Universidad
Pública de Navarra
Nafarroako
Unibertsitate Publikoa

Escuela Técnica Superior de Ingenieros Industriales y Telecomunicación

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Máster

CONTROLADOR DE ELASTICIDAD BASADO EN TÉCNICAS DE CLASIFICACIÓN BINARIA

Aitor González de Mendívil Grau

DIRECTOR

Dr. Federico Fariña Figueredo

CODIRECTOR

Dr. José R. González de Mendívil

Pamplona-Iruña

18 de junio de 2017

Controlador de Elasticidad Basado en Técnicas de Clasificación Binaria

Aitor González de Mendívil Grau

19 de junio de 2017

Índice general

Índice de figuras

Índice de tablas

Resumen

En esta memoria se presenta el Trabajo Fin de Máster en Ingeniería Informática realizado durante el curso 2016-2017.

La motivación para realizar este trabajo se debe a la importante atención que se está poniendo en estos últimos años al desarrollo y despliegue de aplicaciones distribuidas en plataformas Cloud tanto públicas como privadas. La creciente demanda de desarrollo de aplicaciones en el Cloud se debe principalmente al modelo de 'pay-as-you go'. Básicamente consiste en pagar por la infraestructura que se emplea en cada momento. Lo atractivo de este modelo es que las complicaciones derivadas de la compra, instalación y mantenimiento de una infraestructura dedicada a las aplicaciones de la empresa se dejan a cargo de los proveedores de infraestructua (IaaS).

Una de las principales ventajas de utilizar una infraestructura externa es la posibilidad de aprovisionar los recursos bajo demanda de las necesidades de la aplicación o servicio. De esta forma los propietarios del servicio pueden de manera ideal escalar (scale-up or scale-down) los recursos asignados al servicio para mantener los criterios de calidad de servicio de la aplicación (QoS).

Bajo el contexto anterior aparece el concepto de *Elasticidad de un servicio*, término que también es conocido como aprovisionamiento dinámico o planificación de capacidad dinámico. En el trabajo de revisión que se presenta en esta memoria se analizan los aspectos anteriores y se hace una revisión de algunas técnicas utilizadas para controlar y gestionar la elasticidad de los servicios.

Además también se incluye como parte del trabajo una propuesta de controlador para gestionar la elasticidad de las aplicaciones en sistemas Cloud. El controlador se basa en un algoritmo de clasificación binaria y una optimización basada en el algoritmo Hill Climbing, cuya aplicación es novedosa ya que no ha sido realizada hasta el momento en el contexto de la elasticidad. Este controlador se ha desarrollado en Matlab y utiliza un modelo de rendimiento basado en la teoría de colas para simular los tiempos de respuesta de una aplicación Cloud.

Organización de la memoria

El resto de la memoria se organiza de la siguiente forma:

En el Capítulo primero se detallan los conceptos básicos sobre la elasticidad en la nube. Se explican las diferencias entre los IaaS, PaaS y SaaS poniendo una atención especial tanto en la infraestrucutra como servicio (IaaS) como en las plataformas como servicio (PaaS)

En el Capítulo segundo se explican las distintas técnicas de elasticidad estudiadas, entre las que se incluyen las basadas en reglas, las basadas en la teoría de colas y las que utilizan modelos de Kriging.

En el Capítulo tercero se detalla el controlador propuesto basado en sistemas de clasificación binaria junto con la evaluación experimental realizada.

Finalmente, en el Capítulo cuarto se presentan las conclusiones junto con los caminos de investigación que abre este proyecto y los posibles trabajos futuros.

Agradecimientos

En primer lugar me gustaría agradecer a mis directores Federico Fariña y José Ramón González de Mendívil toda la guía y ayuda recibida para la realización de este trabajo.

También quiero agradecer a mi familia Aita, Amatxu y Gorka y a mi novia Ruth todo el ánimo y cariño recibido. Sin ellos este trabajo no hubiera salido adelante.

Por último, me gustaría agradecer a todos mis amigos y, en especial, a algunos compañeros de Stratio, que de manera inidirecta me han animado y apoyado en la realización de este proyecto.

Capítulo 1

Elasticidad en la Nube

1.1. Conceptos básicos: IaaS, PaaS, SaaS

La *Computación en la Nube* (*Cloud Computing*) es, simplemente, un paradigma que permite ofrecer servicios de computación a través de una red, que usualmente es Internet. Bajo este modelo, todo lo que puede ofrecer un sistema informático se ofrece como un *servicio*, de modo que los usuarios puedan acceder a los servicios disponibles 'en la nube de Internet' sin tener que preocuparse en la gestión de los recursos que usan. Entre las ventajas potenciales de este modelo destacan: (i) Prestación de servicios a nivel mundial; (ii) Implementación más rápida y con menos riesgos; (iii) Actualizaciones automáticas que no afectan negativamente a los recursos utilizados; y (iv) Contribuye al uso eficiente de la energía. También presenta algunas desventajas como son entre otras: (i) La disponibilidad de las aplicaciones está sujeta a la disponibilidad de acceso a Internet; (ii) La confiabilidad de los servicios depende de los proveedores de servicios en la nube; (iii) La seguridad de la aplicación; y (iv) Escalabilidad a largo plazo. Estos aspectos se pueden ver más detallados en [1].

Los servicios disponibles en el cloud se pueden clasificar en *IaaS* (*Infrastructure as a Service*), *PaaS* (*Platform as a Service*) o *SaaS* (*Software as a Service*) dependiendo de la funcionalidad que ofrecen a través de la red.

- IaaS: La aplicación alojada en el cloud ofrece como su principal funcionalidad una infraestructura basada tanto en máquinas físicas como en máquinas virtuales y la capacidad de la gestión de éstas. En una infraestructura como servicio un usuario podrá crear nuevas máquinas o instancias eligiendo sus configuraciones tanto de hardware (CPU, Memoria, Discos duros y capacidad..) como de software a bajo nivel (sistema operativo o imagen base para la instancia). A menudo se incluyen en los IaaS la posibilidad de crear y configurar una infraestrucura de red para conectar las máquinas. Muchas organizaciones se apoyan en los IaaS tanto cloud como híbridos (parte del hardware es el propio de la organización) para generar y gestionar sus CPDs (Centros de procesamiento de datos o Infraestructura a nivel de hardware) bajo demanda. Ejemplos de IaaS polulares son: Amazon AWS, Opentack, Rackspace, Google Compute Engine from Google Cloud, entre otros.

- PaaS: Una plataforma como servicio ofrece a través de la web un entorno donde poder ejecutar aplicaciones creadas por el usuario, obviando toda la parte de la infraestructura. Así el usuario tiene que focalizar el esfuerzo en el desarrollo de la aplicación. La mayoría de los PaaS ofrecen sistemas de elasticidad tanto vertical como horizontal para las aplicaciones que tengan en funcionamiento. Ejemplos de PaaS son Google App Engine, Heroku, Azure, entre otros.

- SaaS: En un software como servicio el proveedor de la aplicación ofrece directamente una

aplicación online a disposición del usuario. Hoy en día, hay multitud de casos como por ejemplo: Bitnami, Salesforce, Atlassian Cloud, entre otros.

En la figura 1.1 [2] se puede ver la clasificación anterior organizada en diferentes capas según la responsabilidad de cada una de ellas. El proveedor de cloud controla el aprovisionamiento de máquinas virtuales y su asignación a los recursos físicos disponibles. El proveedor de la aplicación, bien directamente o a través de un PaaS, tiene la responsabilidad de asignar los componentes de la aplicación sobre la infraestructura proporcionada por el proveedor de cloud.

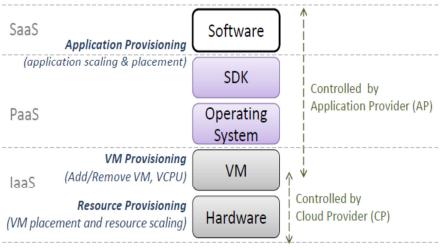

Figura 1.1: Capas con las distintas responsabilidades [2]

1.2. Infraestructura como Servicio: Máquinas Virtuales

Como se ha mencionado anteriormente, los IaaS ofrecen una infraestructura de hardware que puede ser tanto física como virtualizada. Debido a que la estructura de los CPDs suele estar formada por máquinas físicas de grandes recursos, es comprensible que se recurra a sistemas de virtualización para la gestión de todos ellos. Los sistemas de virtualización ofrecen una flexibilidad a la hora de crear máquinas y asignar recursos que no se puede conseguir directamente con el hardware. Además, el proveedor de infraestructura puede así ofrecer distintas imágenes base para que el usuario las aproveche y no tenga que gastar más recursos en proveer la infraestructura. En ese sentido hay una gran ventaja a la hora de disponer de máquinas virtuales ya preparadas con instalaciones de SOs, instalaciones de paquetes de software o middleware (SDK) con el consiguiente ahorro de tiempo a la hora de la gestión y administración de estos sistemas. La mayor parte de hypervisores de máquinas virtuales (MV) ofrecen un api con métodos para añadir/eliminar MVs y otras funcionalidades más complejas como la posibilidad de migrar MVs de un sistema físico a otro a través de la red manteniendo la imagen de estado de ejecución de la MV. En algunos sistemas se ofrece una replicación sincrónica con objeto de aumentar la disponibilidad de las MVs.

Uno de los problemas que deben resolver los proveedores de Iaas es la asignación de MVs a los recursos físicos disponibles con el objetivo de minimizar costes propios y consumo de energía. Sobre la demanda externa de MVs deben minizar el número de recursos físicos pero manteniendo la calidad de servicio suministrada (caractersticas de tiempos de ejecución de CPU, tamaño y tiempo de memoria, latencias y ancho de banda de la red, etc).

Los tres problemas típificados a nivel de IaaS en relación al aprovisionamiento dinámico [2] son:

- Escalado de MVs: determinar el incremento o decremento en el número de MVs y sus recursos asociados como CPU virtual, memoria, etc.
- Emplazamiento de MVs: determinar dónde deben ubicarse las MVs, o asignación de recursos a las MVs.

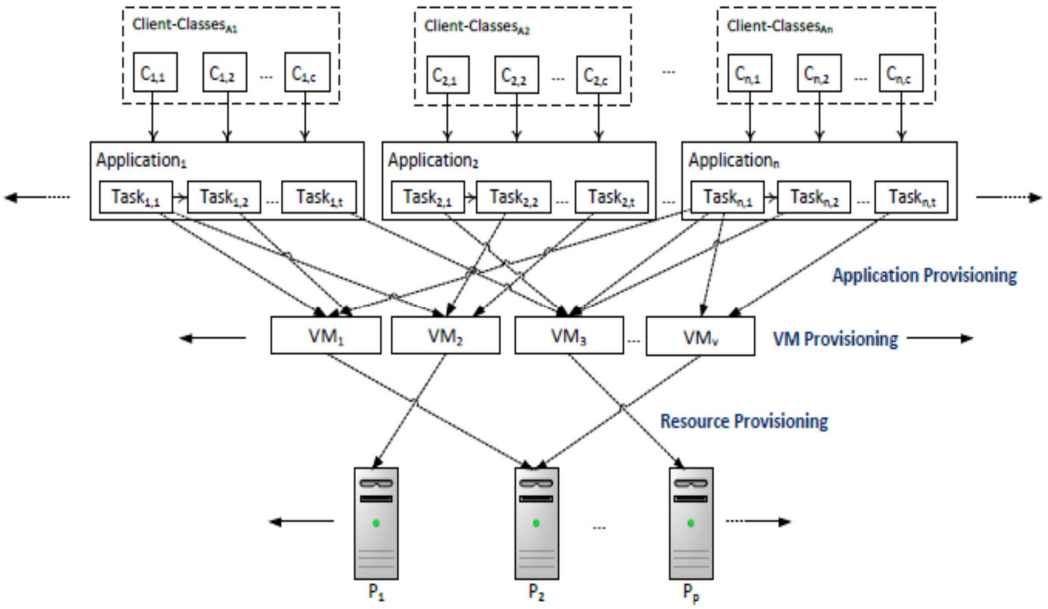

Figura 1.2: Diferentes tipos de aprovisionamiento de recursos en la Nube [2]

- Escalado de Recursos: determinar el incremento o decremento en el número de recursos operativos, cuáles MVs deben ejecutarse y utilizarse.

Esos problemas anteriores son problemas típicos de optimización en función de ciertos objetivos a cumplir. En la figura 1.2 se presenta de forma esquemática los diferentes niveles de asignación de recursos [2].

1.3. Plataformas de despliegue: PaaS

De forma muy escueta, un PaaS permite a los desarrolladores centrarse en el desarrollo de su aplicación y sus componentes de base, ya que la plataforma debe soportar el ciclo de vida del servicio desarrollado. El PaaS puede verse como un medio que ofrece a los desarrolladores el soporte de software, SOs, red etc, necesarios para el desarrollo y ejecución del servicio.

Sin embargo, las prestaciones que se desean para la aplicación que se desarrolla en el PaaS tienen que estar consideradas en las fases de diseño del mismo y deben satisfacerse cuando el sistema está en su fase de consolidación. Los objetivos de estas prestaciones suelen venir en acuerdos de servicio (SLAs) que especifican entre otros aspectos los tiempos de respuesta de las operaciones y sus límites, la productividad del sistema, costes máximos y mínimos tolerados, y la disponibilidad del servicio. Por otra parte, la ventaja potencial del cloud, en general, es la visión que ofrece de que los recursos se pueden establecer de manera dinámica a las aplicaciones para que en todo momento la promesa de 'pagar por lo que usas' sea una realidad.

Por todo ello, los PaaS, como plataformas donde el usuario va a desplegar sus aplicaciones, deben centrarse en resolver una serie de problemas para el correcto funcionamiento de éstas, de

forma que se permita al usuario centrarse en la aplicación y olvidarse de la plataforma. Los retos a los que se tienen que enfrentar los PaaS actuales son los siguientes:

(i) Elasticidad: tanto horizontal como vertical. Las aplicaciones deben consumir sólo los recursos necesarios en cada momento de todos los que están disponibles en la plataforma;
(ii) Despliegue: las aplicaciones deben poder desplegarse de una forma sencilla e intuitiva;
(iii) Tolerancia a fallos: las aplicaciones deben estar disponibles a pesar de que ocurran fallos en los nodos que forman parte de la plataforma;
(iv) Persistencia de Datos: ciertas aplicaciones requieren de persistencia de datos para su funcionamiento.
(v) Monitorización: Las plataformas deben de satisfacer los requerimientos de monitorización de las aplicaciones, ya sea para uso propio (escalado) o para uso del usuario;
(vi) Coste: Las plataformas utilizan un método de cobro denominado 'Pay as You go', caracterizado por incluir sólo los costes de los recursos de los que está haciendo uso la aplicación en un momento dado. Esto pone en evidencia la importancia de la elasticidad de las aplicaciones con el objetivo de ahorrar costes.

Para simplificar el desarrollo y el despliegue de las aplicaciones que se ejecutan sobre los PaaS, éstos están recurriendo a ofrecer estándares basados en *contenedores* (no confundir con sistemas de virtualización) que junto con la ayuda de las arquitecturas basadas en microservicios son capaces de satisfacer algunos de los requisitos mencionados previamente.

Así mismo, el IaaS es vital como capa en el nivel inferior de los PaaS, permitiendo una correcta gestión de los recursos físicos (o virtualizados) sobre los que se sustenta la plataforma de despliegue. Según [2] (ver la figura 1.2) los problemas de asignación dinámica de recursos a nivel del PaaS son básicamente:

- Escalado de la Aplicación: determinar el incremento o decremento en el número de unidades componentes de la aplicación (añadir réplicas de procesos o capas en la aplicación)
- Emplazamiento de la Aplicación: determinar el lugar de las unidades de la aplicación sobre las MVs que utiliza.

Algunos autores [3] indican que la aceptación de llevar a cabo los desarrollos sobre la nube está directamente relacionada con las prestaciones que se obtienen de las aplicaciones y con el cumplimiento de los acuerdos de servicio que se ofrecen. Esto es realmente importante para las plataformas PaaS puesto que trabajan directamente con los proveedores de las aplicaciones. El middleware de ejecución de los PaaS afecta también al rendimiento de la aplicación que despliegan y controlan y este hecho, dificulta el modelado de las prestaciones de las aplicaciones sobre todo cuando se debe controlar su escalabilidad.

En cualquier caso, utilizar un PaaS tiene ventajas sobre los IaaS ya que los PaaS permiten que los esfuerzos de un equipo u organización estén orientados al desarrollo de las aplicaciones, es decir, a su modelo de negocio, obviando los problemas que surgen del aprovisionamiento de la infraestructura.

1.4. Arquitecturas basadas en Microservicios

Las aplicaciones tradicionales, usualmente están formadas por un núcleo en el que se encuentra la lógica de negocio implementada por módulos. Envolviendo este núcleo existen interfaces que se

comunican con el mundo exterior, bases de datos, consumidores/productores de mensajes, apis, interfaces de usuario etc...

Esta arquitectura, a pesar de tener unos componentes claramente diferenciados y por lo tanto ser modular, se empaqueta, se entrega y se despliega como un único componente. De ahí que se denomine a este tipo de arquitecturas como monolíticas. Por ejemplo, una aplicación backend de java que expone una api y una interfaz web se empaqueta formando un archivo WAR, que se despliega en un servidor Tomcat para funcionar. Este tipo de aplicaciones son sencillas de desarrollar, testear, versionar y desplegar (sólo es necesario mantener un archivo) Son sencillas incluso de escalar gracias a los balanceadores de carga (siempre que la aplicación lo permita). No obstante, este tipo de aplicaciones tienden a crecer con el tiempo, convirtiéndose en verdaderos monstruos complejos imposibles de mantener. Resolver los errores y añadir funcionalidad se vuelven cada vez más complicados y costosos, entrando en una espiral que se retroalimenta. Además, con el continuo crecimiento, la velocidad de arranque de la aplicación se ralentizará, y la integración continua será más difícil de llevar a cabo.

Este tipo de aplicaciones y sus dificultades añadidas, ha llevado a buscar nuevas arquitecturas basadas en sistemas más sencillos, que se puedan componer fácilmente, que atajen toda la complejidad mencionada. Una de esas arquitecuras, que es tendencia actual, se denomina *Arquitectura basada en Microservicios.*

La arquitectura basada en microservicios [6] consiste en la división de una aplicación compleja en pequeñas aplicaciones que se encargarán de unas funcionalidades concretas y bien definidas. Por ejemplo, una tienda online se podría descomponer en los siguientes microservicios según la funcionalidad: (i) gestión de usuarios (ii) gestión de stocks (iii) pagos (iv) facturación (v) notificaciones. De esta forma, cada uno de los microservicios puede ser implementado y desplegado de una forma diferente. En caso necesario, dependiendo de las prestaciones, podrían ser replicados, con lo que se ayuda en los procesos de escalabilidad. Además, no sería necesario tener una única base de datos para toda la aplicación, sino que cada uno de los microservicios puede incorporar la base de datos (y el esquema de datos) más adecuado para sus necesidades. Otra ventaja es que el desarrollo de cada microservicio es fácilmente paralelizable. No obstante, no todo son ventajas. La gestión de la configuración, y la gestión de las versiones en aplicaciones distribuidas de este tipo de arquitecturas se vuelve más compleja, al igual que los procesos de testing y CI (Continuous Integration). Finalmente, es importante destacar que la comunicación entre los diferentes microservicios debe estar claramente definida antes del comienzo del desarrollo de éstos, si no, los fallos que surgirían a la hora de la integración de los componentes provocarían perder muchas de las ventajas que proporcionan este tipo de arquitectura. Empresas tan importantes como Netflix, Amazon o Ebay basan sus sistemas de producción en microservicios.

Las tecnologías basadas en contenedores como pueden ser Docker, LXC o Rocket son muy útiles a la hora de afrontar los problemas que surgen en una aplicación distribuida basada en microservicios, permitiendo aislar los distintos microservicios, y sus tecnologías asociadas, y ofreciendo un sistema de gestión que es controlable mediante un API o un CLI.

1.5. El concepto de Elasticidad

En los apartados anteriores hemos hablado informalmente de dos conceptos fundamentales para los servicios desplegados en la nube como son la *escalabilidad* y la *elasticidad* de los servicios, sin tener una definición precisa de tales conceptos. En este apartado se intenta explicar mejor estas propiedades de los servicios en la nube siguiendo las ideas expuestas en [4]. Según Herbst [4]

Elasticidad*: es el grado por el cual un sistema es capaz de adaptarse a los cambios en la carga de trabajo que soporta mediante el abastecimiento y desabastecimiento de recursos de cómputo de forma autónoma, de tal manera que en cada punto del tiempo los recursos disponibles encanjan con la demanda de recursos real tan cerca como sea posible.*

A partir de esta definición se observa que la escalabilidad es un requisito indispensable para que un sistema pueda tener un cierto grado de elasticidad. Recordemos que un sistema es escalable si sus prestaciones no se degradan de manera significativa cuando la carga del sistema aumenta. De todas formas, la noción anterior de escalabilidad también es una definición informal. Hasta el año 2000 no se propuso una definición más formal de está noción. Jogalekar en [5] propone que para definir la escalabilidad de un sistema hay que definir ciertas variables primarias de 'escalado' sobre las cuales puede ofrecerse una métrica para este concepto. Por ejemplo, en un sistema distribuido que utilice una base de datos, el número de réplicas de la base de datos puede ser tal factor de escalado k, de tal manera que otras variables que son el objetivo del sistema como puede ser la productividad (número de operaciones por segundo) o el tiempo de respuesta de las operaciones, puedan ser definidas en función del parámetro de escalado. De esta forma la métrica que indica la escalabilidad del sistema puede calcularse con la siguiente estrategia:

1. Se calculan las siguientes cantidades a partir del factor de escalado k:
   $\lambda(k)$: productividad en respuestas por segundo.
   $f(k)$: 'valor' medio de cada respuesta calculado desde la definición de calidad de servicio para el factor k.
   $C(k)$: 'coste' de la configuración con factor k, expresado como el coste de ejecución por segundo.

2. Se calcula la productividad como $F(k) = \lambda(k) \cdot f(k)/C(k)$

3. El factor de escalado para dos configuraciones del parámetro de escala k_1 y k_2 se define como la proporción de sus productividades: $\Phi(k_1, k_2) = F(k_2)/F(k_1)$

De esta forma un sistema es escalable si su productividad va acorde con los costes y el valor de Φ se mantiene por encima de la unidad (o próximo a la unidad, por ejemplo 0,8).

Hay que tener en cuenta que la definición de Jogalekar es independiente de si hay un cambio de configuración en el sistema o no. Por ejemplo, se puede mantener una configuración de recursos fija y estudiar la escalabilidad para un factor de escala que puede ser el número de operaciones admitidas concurrentemente dentro del sistema. Por otro lado, lo que diferencia el concepto de escalabilidad de la elasticidad es que en la definición de escalabilidad no se tiene encuenta ningún aspecto temporal como por ejemplo en qué momento deben realizarse ciertas acciones de escalado.

No obstante, para conseguir que un sistema sea escalable se require un esfuerzo en el diseño del sistema; en otras palabras, hay que utilizar diferentes técnicas que permitan que el servicio no se degrade rápidamente en cuanto aumente la carga de trabajo. Estas técnicas son bastante conocidas, por ejemplo: evitar bloqueos, repartir la ejecución de tareas entre distintos nodos, utilizar mecanismos de 'caching', repartir los datos entre los nodos ('sharding') para favorecer el reparto de las operaciones, utilizar protocolos de comunicación del tipo publicador/subscriptor, replicar los componentes de una aplicación.

En la actualidad, debido a la influencia del uso de MVs y contenedores, se han hecho populares dos técnicas de escalado:

- Horizontal. El escalado horizontal consiste en la creación o destrucción de componentes de la aplicación con el objetivo de adecuar la carga de éstos, consiguiendo de esa forma que la aplicación cumpla con el SLA (Service Level Agreement)

- Vertical. Consiste en aumentar o reducir los recursos de la máquina que está sirviendo una aplicación con el mismo objetivo de que satisfaga el SLA. Aún existiendo sistemas de virtualización que permiten la posibilidad de aumentar los recursos asignados, este tipo de escalado es más complejo de llevar a cabo.

Como observamos, la escalabilidad horizontal se consigue creando o destruyendo componentes de la aplicación, normalmente mediante técnicas de replicación y distribución de carga. Realmente el escalado horizontal supone un cambio en la configuración del sistema y por lo tanto, en los recursos que se emplean en el mismo. Pero para que el escalado horizontal permita convertir al sistema en un sistema elástico se necesita un proceso de adaptación que permita en cada momento tomar la decisión de qué componentes deben duplicarse o no y en qué cantidad.

En términos más generales, Herbst [4] propone que para evaluar la elasticidad hay que tener en cuenta los siguientes aspectos:

- Escalado autónomo: cuál es el proceso de adaptación que se utiliza.
- Dimensiones de elasticidad: cuál es el conjunto de tipos de recursos que son escalados como parte del proceso de adaptación.
- Unidades de los recursos que escalan: para cada tipo de recurso, cuál es la unidad de variación de la cantidad de recursos que pueden asignarse.
- Límites del escalado: para cada tipo de recursos, cuál es el límite superior e inferior de la cantidad de recursos que pueden ser asignados.

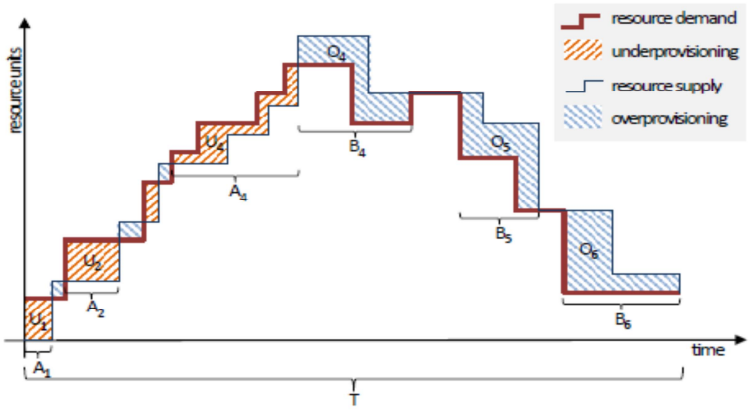

Figura 1.3: Comportamiento de un sistema elástico [4]

En el proceso de adaptación puede haber múltiples tipos de recursos que pueden escalarse ('up' o 'down'). Cada tipo de recurso puede verse como una dimensión diferente en el proceso de adaptación con sus propias políticas de elasticidad. Normalmente, el abastecimiento de estos recursos es de naturaleza discreta. El sistema funcionando con un mecanismo de elasticidad puede estar en un momento dado en un estado sobre-abastecido ('overprovisioned') o bajo-abastecido ('underprovisioned') en comparación con un valor ideal de abastecimiento óptimo que cumple precisamente con la calidad de servicio exigida.

De esta forma, los dos aspectos centrales del mecanismo de adaptación pueden ser:

(i) *Velocidad*; el tiempo que tarda el sistema de pasar de un estado bajo-abastecido a un estado óptimo o sobre-abastecido, y viceversa, el tiempo que tarda el sistema de pasar de un estado sobre-abastecido a un estado óptimo o bajo-abastecido.
(ii) *Precisión*, la desviación de la cantidad actual de los tipos de recursos asignados respecto a la demanda de recursos real en ese momento.

En la figura 1.3 tomada de [4] se muestra en el eje vertical las unidades de un tipo de recurso escalable y en el eje horizontal el tiempo. La demanda real de ese tipo de recurso en cada momento se representa por la línea negra. Como consecuencia del proceso de adaptación, el sistema puede encontrarse en estados sobre-abastecidos (en azul) o bajo-abastecidos (en rojo). Las duraciones de estos estados y sus valores acumulados pueden utilizarse para definir métricas sobre el grado de velocidad y precisión del sistema, y por tanto, de su grado de elasticidad [4].

Capítulo 2

Técnicas de Elasticidad

2.1. Métodos generales de autoescalado

En la literatura se han presentado varios trabajos que representan el estado del arte de la elasticidad en la nube desde diferentes puntos de vista. El trabajo de Shoaib y Das [2] presenta una taxonomía de las técnicas que se utilizan en el abastecimiento de recursos desde una perspectiva orientada a las prestaciones del servicio.

Su clasificación se basa en responder a ciertas preguntas relativas a

(i) quién es el responsable del aprovisionamiento;
(ii) cuáles son los aprovisionamientos de recursos, indicando las técnicas de escalado, los problemas que se deben resolver y las políticas del emplazamiento de los recursos;
(iii) cuándo se realiza la acción de escalado, básicamente si la técnica es reactiva o predictiva;
(iv) qué tipos de recursos y sus particularidades intervienen en el escalado;
(v) dónde se toman las decisiones de escalado;
(vi) cómo se realiza en particular la toma de decisiones y qué técnicas y tecnologías están asociadas a todo el proceso de aprovisionamento dinámico.

En la figura 2.1 se presenta la clasificación prouesta por Shoaib y Das [2]. En este capítulo nos vamos a centrar principalmente en las técnicas de autoescalado, las cuáles estarían englobadas dentro de la toma de decisiones en la figura 2.1. El trabajo de Lorido–Botran [7] analiza principalmente las técnicas de autoescalado que se pueden aplicar para dotar de elasticidad a los servicios en la nube.

Las técnicas de autoescalado encajan dentro del modelo tradicional MAPE-K propuesto para los sistemas autónomos. Este modelo está descrito en el informe de IBM del 2005 [8] relativo a *Autonomic Computing*.

En este modelo, el sistema (ver figura 2.2) autónomo tiene un lazo principal que consiste en

(i) Monitorización (M) de parámetros del servicio;
(ii) Análisis (A) de los valores de la señales para la detección de algún tipo de anomalía;
(iii) Planificación (P) de las acciones a realizar en función de las decisiones tomadas en el análisis;
(iv) Ejecución (E) de las acciones siguiendo la planificación prevista;
(v) el proceso MAPE se guarda en una base de datos de conocimiento (K) para análisis futuros o para crear nuevos comportamientos.

En un sistema elástico el subsitema de autoescalado tiene relación con las fases de análisis y de planificación puesto que el autoescalado utiliza la información monitorizada para realizar

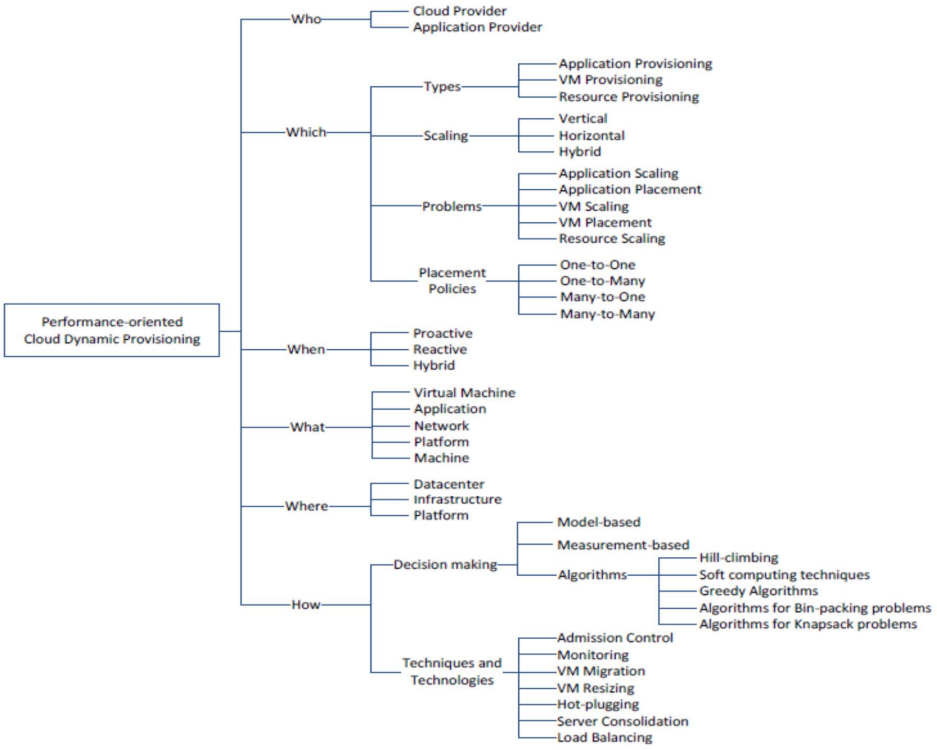

Figura 2.1: Una clasificación del aprovisionamiento dinámico en la nube [2]

estimaciones futuras de las necesidades de los recursos y realizar un planificación de la acciones de modificación, por ejemplo, crear o destruir una máquina virtual, o duplicar un contenedor para tener replicada la funcionalidad de un componente. Así, mediante la distribución de carga las prestaciones del sistema pueden mejorar por estas acciones de escalado horizontal.

Lorido–Botran et. al. [7] proponen clasificar las técnicas de autoescalado en función de la técnica particular que se utiliza en el subsistema de autoescalado. No obstante, esta clasificación es una aproximación ya que hay un gran número de combinaciones que aparecen en la literatura.

1. Reglas basadas en umbrales ('Threshold-based rules'). Consiste en utilizar directamente las medidas que se monitorizan de los componentes del servicio para definir reglas de acción sobre los componentes cuando las medidas superan ciertos valores fijados por el desarrollador del servicio. La ventaja de esta técnica es su sencillez de implantación y su principal desventaja consiste en que los resultados obtenidos por las acciones no son excesivamente precisos provocando sobredimensionamientos que no son realmente necesarios.

2. Aprendizaje mediante reforzamiento ('Reinforcement learning'). En estas técnicas los estados del sistema obtenidos mediante la monitorización de los componentes se utilizan para ir aprendiendo un modelo, normalmente basado en Modelos probabilistas de Markov. Este modelo indica cuál es la acción que hay que realizar sobre el sistema, en función del estado actual,

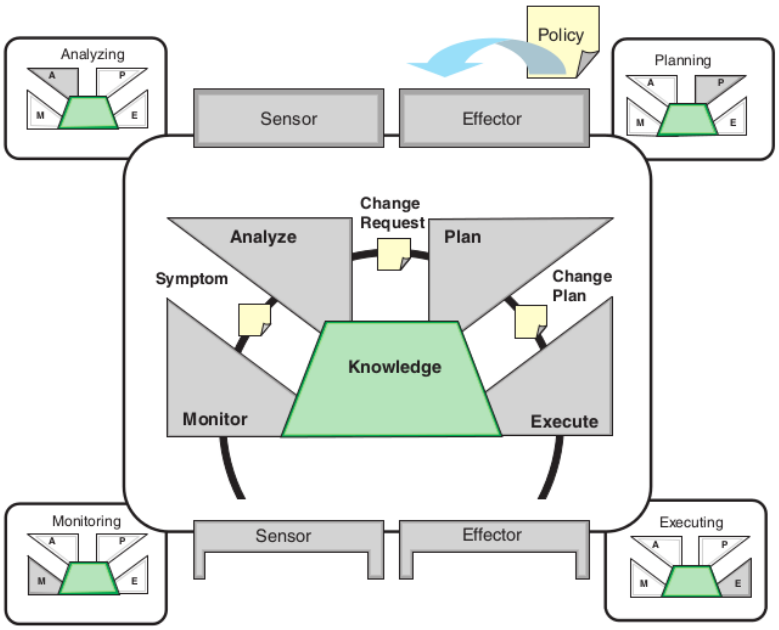

Figura 2.2: Modelo MAPE-K [8]

para ir obteniendo el mejor ajuste con respecto a la QoS deseada. La función de utilidad o de penalización se emplea como método de búsqueda de dicha acción. Su principal ventaja es que las acciones de escalado (normalmente escalado horizontal) son bastante precisas y no generan aprovisionamientos innecesarios pero su desventaja es el tiempo que se debe dedicar al aprendizaje del modelo. Por otra parte, se debe tratar con cuidado la explosión de dimensionalidad que puede suceder en función del gran número de parámetros que definen el estado del sistema.

3. Teoría de Colas ('Queueing theory'). La teoría de colas ha sido la teoría dominante en el estudio de las prestaciones de los sistemas informáticos. Mediante esta teoría se pueden establecer modelos aproximados que permiten obtener los valores relativos al tiempo de respuesta o a la productividad que se obtiene en un sistema informático conociendo de antemano los valores de los tiempos de servicio de los componentes y la carga de trabajo sobre los mismos. El uso de estas técnicas consiste en establecer un modelo de colas para el servicio que permita predecir su comportamiento cuando ciertas condiciones sobre la carga del sistema cambia. Con dicho modelo se puede predecir si los parámetros de QoS pueden violarse en la configuración actual. Teniendo disponibles otros modelos para otras configuraciones se puede realizar una búsqueda de la mejor configuración para una carga de trabajo determinada. Este modelo tiene como ventaja la precisión que se puede obtener con las acciones de escalado definidas por los cambios de configuración y como principal desventaja las dificultades de obtener de forma precisa los valores de los parámetros que definen el modelo de colas utilizado.

4. Teoría de Control ('Control theory'). Si observamos el modelo MAPE-K, éste no difiere mucho de los sistemas de control típicos que se pueden encontrar en la mayoría de los sistemas automáticos o electrónicos. La teoría de control establece modelos matemáticos (basados en ecuaciones diferenciales) donde se estudian los lazos de retroalimentación para definir la señal de control que debe generarse sobre el sistema para seguir una señal de referencia. Fundamental a esta teoría es el estudio de los sistemas adaptativos que consisten en definir

un modelo matemático del sistema bajo control para definir la mejor acción de control. Por este motivo, algunas de las técnicas de autoescalado que se utilizan en sistemas elásticos se basan en esta teoría de control. Las ventajas e inconvenientes que tiene esta técnica son similares a los obtenidos mediante la teoría de colas. Por otro lado, la mayor parte de los trabajos que emplean esta técnica también hacen uso de los modelos de colas como la planta a controlar. En la figura 2.3 se muestra un sistema típico de control con retroalimentación.

5. Análisis de series temporales ('Time series analysis'). Aunque el estudio de las series temporales no es una técnica propiamente dicha de autoescalado, se incluye en la clasificación por la importancia que tiene en los sistemas desarrollados puesto que es la base para realizar predicciones futuras sobre el comportamiento del sistema a partir del conocimiento de su comportamiento en tiempos del pasado. La base del análisis temporal consiste en resolver una ecuación funcional del tipo $y_t = f(y_{t-k}, x_{t-k})$ donde y_t es la predicción para el valor de la señal $y(t)$ en el instante t, y_{t-k} son los valores pasados de la señal, y x_{t-k} son ciertos parámetros de ajuste.

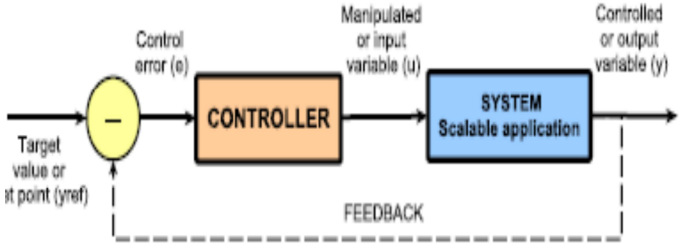

Figura 2.3: Lazo de retroalimentación en un sistema de control [7].

2.2. Técnicas de autoescalado basadas en reglas estáticas

Dentro de las técnicas de autoescalado que se aplican para la construcción de sistemas elásticos existen los métodos basados en reglas estáticas. Estos métodos se caracterizan por predefinir una serie de reglas, que en base a una serie de valores tomados del sistema redimensionarán (scale-up/ scale-down) la configuración del servicio con el objetivo de cumplir con los valores de calidad de servicio establecidos en el SLA con el menor coste posible.

Este tipo de técnicas, basadas en reglas estáticas son muy populares entre los clientes cloud (Amazon EC2 por ejemplo) por sus simplicidad, no obstante, la toma de medidas y la definición de los umbrales correctos para estas reglas necesitan un amplio conocimiento del funcionamineto de la carga de la aplicación y del diseño de la misma.

2.2.1. Descripción de la técnica

La mayor parte de los sistemas cloud ofrecen como parte de sus servicios el escalado basado en reglas, que está orientado a la escalabilidad de instancias o máquinas virtuales, no obstante,

más adelante veremos cómo este mismo concepto se puede utilizar para escalar servicios en una aplicación distribuida.

Las reglas para el escalado, ya sea horizontal o vertical se estructuran de una forma igual o similar a la mostrada en la figura 2.4. Cada regla consta de dos partes, la condición y la acción que será ejecutada cuando se cumpla la condición. La condición necesita del uso de métricas de rendimiento obtenidas del sistema, como pueden ser, carga de cpu, carga de memoria, I/O de red o de disco, o de métricas propias del servicio (llegada de peticiones por segundo, número de peticiones deshechadas, etc.). Cada una de estas métricas tiene un umbral superior $thrU$ e inferior $thrL$. Cuando la condición sea cierta durante un cierto tiempo ($durU$ o $durL$) entonces la acción será ejecutada. En un escalado horizontal, se deberá definir un número fijo de s máquinas virtuales con las que ampliar o reducir el sistema, mientras que para realizar un escalado vertical, s se referirá a la cantidad de un recurso hardware que se deberá aumentar o reducir, como puede ser memoria RAM, número de núcleos de la CPU, tamaño en disco, etc. El proceso de escalado puede producir un periodo de *downtime* definido por inU o inL en el que el sistema de reglas no debería actuar con el fin de volver a otorgar estabilidad al sistema.

$$\begin{aligned} &\text{if } x_1 > thrU_1 \text{ and/or } x_2 > thrU_2 \text{ and/or } \ldots \\ &\quad \text{for } durU \text{ seconds then} \\ &\quad\quad n = n + s \text{ and} \\ &\quad\quad \text{do nothing for } inU \text{ seconds} \end{aligned} \tag{1}$$

$$\begin{aligned} &\text{if } x_1 < thrL_1 \text{ and/or } x_2 < thrL_2 \text{ and/or } \ldots \\ &\quad \text{for } durL \text{ seconds then} \\ &\quad\quad n = n - s \text{ and} \\ &\quad\quad \text{do nothing for } inL \text{ seconds} \end{aligned} \tag{2}$$

Figura 2.4: Estructura de reglas estáticas [7]

Un ejemplo de regla podría ser el siguiente: *añadir una nueva instancia de la imagen base centOS cuando la carga de CPU media de las imágenes disponibles en el sistema supere el 70 % durante 2 minutos y desactiva el sistema de reglas durante 10 minutos.*

En realidad, el sistema de reglas puede comportarse de dos formas:

- Reactiva: Son aquellas que reaccionan después de una serie de condiciones obtenidas de forma directa, tal y como hemos visto en el ejemplo anterior. La métrica de CPU es una métrica obtenida directamente del sistema.
- Proactiva (predictiva): Este tipo de reglas incluyen en sus condiciones una predicción de ciertas métricas de forma que las acciones se adelanten a los acontecimientos. Generalmente, a la hora de implementar este tipo de reglas se utilizan diferentes algoritmos de predicción basados en series temporales.

Hay que tener en cuenta varias cuestiones que son importantes. La primera hace referencia a que la acción de escalado, por ejemplo añadir una nueva máquina virtual, tiene una duración que

no es despreciable (unos cuantos minutos). Por este motivo, los sistemas reactivos tienen un peor comportamiento que los sistemas proactivos, ya que estos últimos (si la predicción de la métrica es precisa) se adelantan y son capaces de compensar el retraso que produce la acción de escalado correspondiente.

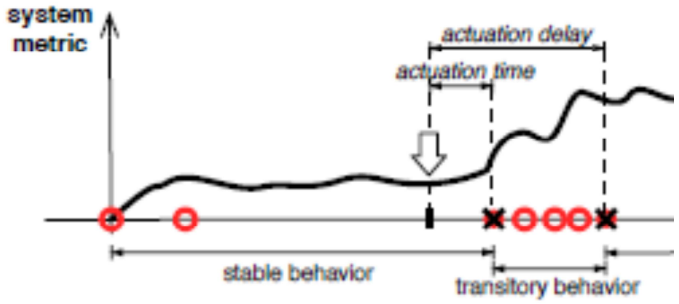

Figura 2.5: Retraso en los efectos de la acción de las reglas

En la figura 2.5 podemos observar el efecto del retardo que se produce cuando la acción es reactiva. El tiempo de actuación refleja que a partir de un cambio en alguna métrica, por ejemplo la carga de entrada, pasa un tiempo (definido en la regla) para ejecutar la acción. Pero el efecto de esta acción sobre el sistema no es inmediato como refleja el retardo producido por los efectos de la acción.

La segunda cuestión es que no todas las aplicaciones son tan simples que admiten el que parte de sus componentes instalados en una MV sean duplicados o eliminados sin comprometer la corrección de la aplicación. Finalmente, las reglas deberían trabajar en conjunto y no de forma individual, en el siguiente sentido, un grupo de reglas que afectan a parte de un servicio deberían alcanzar algún tipo de acuerdo para producir cambios de configuración *suaves* para alcanzar los criterios de QoS. Estos aspectos hay que tenerlos en cuenta a la hora de definir y sintonizar un sistema de reglas para un servicio dado.

No obstante, por su sencillez, es más habitual encontrar reglas reactivas. Sin embargo, en [9] se definen unos mecanismos de predicción para la escalabilidad de servicios (en este caso se aplican las reglas de escalabilidad a un nivel de plataforma y no al nivel de infraestructura) que se están ejecutando en el sistema. Para ello, utiliza Docker como motor de contenedores, de forma que un contenedor pasa a ser el 'recurso' que se necesita escalar. Así Kan define en [9] un sistema de escalabilidad horizontal que aplica reglas tanto reactivas como proactivas. El diseño de la arquitectura que propone Kan incluye un balanceador de carga, un sub-sistema de monitorización, un subsistema de aprovisionamiento con un controlador de elasticidad, un repositorio de docker privado y una serie de contenedores Docker con aplicaciones web. En lo que sigue se comentan los aspectos de este trabajo como ejemplo de un sistema basado en reglas.

- Balanceador de Carga: Es la entrada a la aplicación web. Las peticiones que llegan se redirigen repartiendo la carga hacia los diferentes contenedores Docker mediante un HAProxy con keepalived, que mantiene más de una instancia del HAProxy con el objetivo de tener el

sistema en alta disponibilidad[1].

- Monitorización y Aprovisionamiento: La monitorización se realiza obteniendo métricas de los contenedores Docker a través de la API de Docker, y los datos de carga (peticiones por segundo) de la página de estadísticas de HAProxy. Estos datos se almacenan en una base de datos, y serán consultados por el controlador de elasticidad. El sistema de aprovisionamiento también está basado en la API de Docker, manteniendo siempre el número de contenedores definido por el Controlador de Elasticidad. Cuando un contenedor se cae de forma repentina el sistema de aprovisionamiento se encarga de intentar reiniciarlo o de ejecutar uno nuevo en caso de que no sea posible.

- Repositorio Docker privado (Docker Registry): Es un servidor sin estado y escalable que permite almacenar imágenes de Docker con el objetivo de poder ejecutar contenedores de una forma sencilla desde cualquier máquina que tenga acceso al servidor. Este servidor está también disponible como una imagen de Docker.

No obstante, para este trabajo la parte que más interés provoca es el controlador de elasticidad explicado en la siguiente subsección.

2.2.2. Controlador de Elasticidad

Como ya se ha mencionado, el control de elasticidad propuesto por [9] incorpora tanto modelos reactivos como proactivos.

- Modelo proactivo: Se encarga de estimar la carga de llegada (peticiones por segundo) después de un periodo de tiempo ΔT y transformar esa información en el número de contenedores. Para predecir la carga en un futuro cercano hace uso de un método ARMA (autoregressive moving average) de segundo orden cuya ecuación es:

  $$y_{t+1} = \beta \times y_t + \gamma \times y_{t-1} + (1 - (\beta + \gamma)) \times y_{t-2}$$

  Los parámetros β y γ se aprenden mediante técnicas estadísticas. Utilizando esta fórmula el resultado que muestra es realmente acertado, no obstante habría que analizar el periodo de tiempo máximo en el que la diferencia con la realidad es aceptable con el objetivo de predecir la carga con la mayor anterioridad posible para dar tiempo a que los docker se pongan en funcionamiento y se minimize el retraso en su actuación. Así, $\Delta T \geq \text{Tproc} + \text{Tdocker}$ donde el Tproc es lo que tarda el motor en calcular el número de dockers, y Tdocker el tiempo que se tarda en arrancar o eliminar un contenedor docker. Para calcular el número de contenedores se asume que se conoce el número de peticiones por segundo f que puede manejar un único contenedor de forma que

  $$N_{proact}(t) = \lambda(t)/f$$

  donde $\lambda(t)$ es la carga predicha para el instante t.

[1] En general, las aplicaciones que se usan en producción no son tan simples como las utilizadas en este ejemplo, en la que cada contendor docker ejecuta la misma aplicación web, sino que necesitan de sistemas más avanzados de service discovery y balanceo de carga.

El cálculo del número de dockers lo realizan mediante la fórmula anterior que es muy básica para una aplicación que se replica con el mismo tipo de contenedor como en el ejemplo que proponen, no obstante se debería haber realizado un análisis más exhaustivo sobre aplicaciones más complejas y, a la vez, tan habituales como las de tres capas (frontal, negocio, datos). De esta forma sobre su modelo se podrá proponer $N^k_{proact}(t) = (\lambda(t)/f^k) \times \alpha_k$, donde k es la capa correspondiente, f^k el flujo máximo para un contenedor de la capa k, y α_k es un factor de escalado para la capa k. En la siguiente sección de este capítulo se abordan técnicas más elaboradas para este tipo de aplicaciones.

- Modelo reactivo: Utiliza un sistema de una única regla en la que se establece un umbral superior de $T_{up} = 0{,}8$ para la utilización de la CPU, con el algoritmo siguiente:

function reactiveScaling()
N:= 0; $N_{reactive}$:= 0;
for container i in running instances N_{inst} **do**
  if $R_i \geq T_{up}$ **then** $N := N + 1$ **endif**
endfor
$N_{reactive}$:= $\lceil N \times (1 - T_{upp})/T_{upp} \rceil$
return $N_{reactive}$
endfunction

En la función anterior R_i es la medida obtenida por el monitor para el valor de la utilización de la CPU del contenedor i.

- Algoritmo completo de escalado: la función anterior responde inmediatamente a la carga que se debe soportar aumentando el número de instancias, pero en el caso de escalar hacia abajo, y con el objetivo de evitar oscilaciones del controlador, se require que durante al menos k periodos el modelo proactivo indique que el número de instancias debe ser menor al que se están empleando en ejecución. El algoritmo completo se presenta a continuación como ejemplo del tipo de algoritmos utilizados en la literatura usando estas técnicas basadas en reglas.

Scaling algorithm
$lasttime$:= 0; (variable global para retrasar el escalado hacia abajo)
function totalScaling(N_{inst}, k)
N_{proact}:= proactiveScaling();
$N_{reactive}$:= reactiveScaling();
if $N_{reactive} > 0$ **then**
  f:= $(\lambda(t) + \lambda(t-1))/2$ (ajustar experimentalmente f)
  $lasttime$:= 0;
  return $N_{reactive} + max(N_{inst}, N_{proact})$
else if $N_{proact} \geq N_{inst}$ **then**
  $lasttime$:= 0;
  return N_{proact}
else if $lasttime \geq k$ then (retrasa el escaldo hacia abajo)
  $lasttime$:= 0;
  return N_{proact}
else
  $lasttime$:= $lasttime + 1$
endif
endfunction

En la función anterior N_{inst} es el número de instancias docker en el periodo actual y k es el número de periodos que deben pasar para que se pueda realizar un escalado hacia abajo, eliminando contenedores sobrantes. El algoritmo anterior no es general y requiere un ajuste experimental. Por ejemplo, el cálculo de f es una aproximación en el caso de que no se haya definido experimentalmente. En realidad no hay ningún soporte teórico que ayude a la realización de este tipo de técnicas basados en reglas ya que se basan en la experimentación de tipo prueba-error.

En los resultados experimentales llevados a cabo asegura que DoCloud presenta un buen grado de precisión y velocidad aunque el tipo de servicio que emplea es realmente simple [9]

2.3. Técnicas de autoescalado basadas en la teoría de colas

Los modelos analíticos basados en la teoría de colas para calcular las prestaciones de los sistemas informáticos han sido exhaustivamente estudiados desde hace años [10] debido principalmente a la existencia de algoritmos sencillos que permiten aproximar con bastante exactitud los resultados del modelo con los resultados experimentales obtenidos sobre los sistemas. Por otro lado, la teoría de colas tiene un soporte matemático basado en la teoría de la probabilidad. En el contexto de su aplicación como base para la definición de una técnica de escalado, el modelo analítico es importante por varias razones:

- Capacidad de aprovisionamiento. Permite a una granja de servidores determinar cuánta capacidad asignar a una aplicación para servir los picos de carga de trabajo.
- Predicción del rendimiento. Permite que se determine el tiempo de respuesta para una carga de trabajo dada una determinada configuración de hardware y software.
- Configuración de la aplicación. Permite que se determinen varios parámetros de la configuración para conseguir las metas de QoS.
- Identificación de cuellos de botella. Permite identificar los componentes que son los responsables de que los tiempos de respuesta no sean aceptables dentro de los parámetros de QoS establecidos.
- Política de peticiones. Permite establecer un límite a la carga de entrada para evitar sobrecargas en el sistema.

La mayor parte de los sistemas modernos son diseñados siguiendo un paradigma de múltiples capas porque permite una aproximación modular. Cada capa ofrece una cierta funcionalidad a su capa predecesora como parte del procesamiento global de una petición.

En la figura 2.6 se muestra un ejemplo de arquitectura en tres capas. Cada petición, si es admitida, pasa por un primer balanceador de carga que la dirige a un servidor concreto y en caso necesario, la petición progresa a la siguiente capa generando varias visitas a dicha capa y posiblemente realizando nuevas peticiones a un base de datos, que suele ser el componente final donde se encuentran los datos persistentes. Cada capa puede estar configurada con varios servidores instalados en contenedores o en máquinas virtuales. Los servidores en las capas intermedias pueden duplicarse puesto que en general son componentes que no guardan estado.

El sistema anterior puede modelarse como una red de colas sencillas. Urgaonkar en [11] propone un modelo como el que aparece en la figura 2.7.

En esta red de colas aparecen las probabilidades p_i para pasar de una cola a otra, donde p_M en la figura toma valor 1. El parámetro Z es característico de un sistema cerrado y es el Think-time del cliente que genera la petición. En este modelo se asume que el tiempo de respuesta medio para las peticiones se calcula con una población de N sesiones concurrentes. Para llevar a cabo el cálculo del

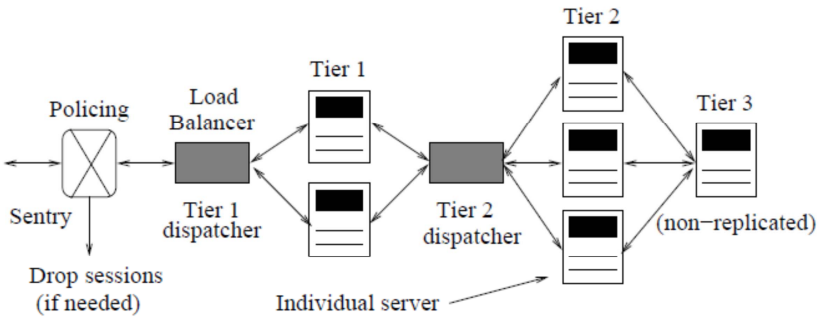

Figura 2.6: Una aplicación estructurada en tres capas [11]

tiempo de respuesta se puede utilizar el algoritmo MVA (Mean Value Analysis) [10] para sistemas cerrados. Este algoritmo está descrito en el Apéndice final de esta memoria. Una de las ventajas de utilizar este algoritmo es que no es necesario conocer las probabilidades p_i, basta con conocer las visitas que una operación realiza a cada una de las colas. Así los parámetros de entrada para el algoritmo son las demandas $D_i = V_i \times S_i$ que se calculan como el producto de la visita a la cola i V_i por su tiempo de servicio medio S_i, el Think-time Z, y obviamente el número de colas $i : 1..M$. Conociendo estos parámetros, el algoritmo calcula el tiempo de respuesta medio y la productividad media. Aunque el algoritmo se presenta para un único tipo de operaciones puede generalizarse para distintas clases de operaciones.

El controlador de la elasticidad basado en este tipo de modelos se basa en lo siguiente: Con la medida de la carga de peticiones de entrada (o su estimación), en este caso N, se utiliza el modelo actual para calcular el tiempo de respuesta medio. Si este tiempo de respuesta no es aceptable, por violar el SLA, entonces se observa sobre los datos del algoritmo cuál es el componente o los componentes responsables del aumento de este tiempo de respuesta, bien mirando la utilización de la colas o los tiempos de respuesta que proporcionan. Llegados a este punto, sobre el mismo modelo se puede generar un nuevo modelo duplicando las colas que se han visto más cargadas. Estas colas, al ser duplicadas, tendrán un menor número de visitas, calculadas como una proporción al número de colas actual que forma el modelo de la capa. Sobre el nuevo modelo se vuelve a calcular tiempo de respuesta medio y en caso de satisfacer el SLA se procede a instalar la nueva configuración. En todo caso, este nuevo modelo, configuración y sus parámetros, se guarda para una futura inspección. En el caso de que ya hubiese una historia previa de modelos almacenados se puede realizar una búsqueda del modelo que económicamente es más rentable para satisfacer el SLA. Este proceso es computacionalmente costoso, sin embargo, los periodos de control de la elasticidad suelen, en general, tener una duración del orden de 5 a 10 minutos como ha sido indicado en otros trabajos.

El proceso anterior sólo es adecuado si se realiza una estimación adecuada de los parámetros que determinan el comportamiento de la red de colas. Estas estimaciones se basan en las medidas de monitorización de los componentes en las capas. Por ejemplo, si queremos calculas V_i se puede

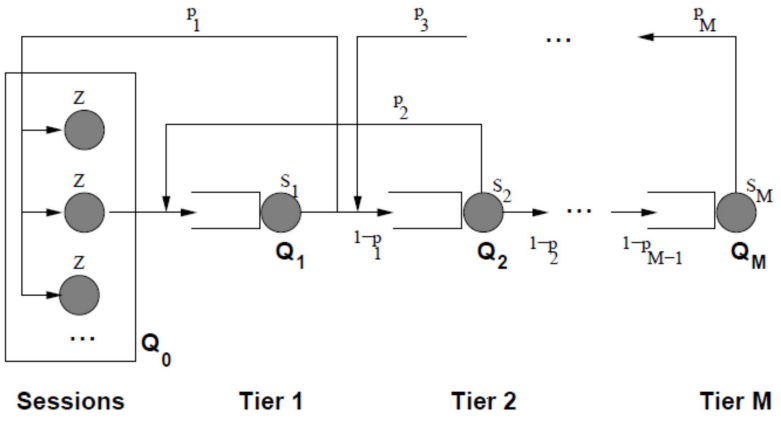

Figura 2.7: Modelo general de una aplicación multicapa utilizando una red de colas simples [11]

aplicar la 'Ley de flujo forzado' que establece que $V_i = \lambda_i/\lambda_{req}$ donde λ_{req} es el número de peticiones servidas por la aplicación global sobre una duración t y λ_i es el número de peticiones servidas por el componente i en ese tiempo. La estimación de los tiempos de servicio S_i es mucho más compleja.

Este problema de la estimación de parámetros para el algoritmo MVA ha sido abordado por Zheng en [12]. Zheng propone hacer un seguimiento continuo de la estimación de los parámetros del modelo utilizando en su caso el denominado Filtro de Kalman.

En la figura 2.8 se presenta un esquema de su propuesta. El algoritmo MVA es simplemente una función no lineal $y = h(x)$ donde y representa el vector que contiene toda la información que se puede obtener del algoritmo y que es potencialmente medible sobre el sistema, y x es el vector que define los parámetros de las colas utilizadas en una determinada configuración. Con las medidas de monitorización z tomadas directamente sobre el sistema se calcula el error $e = z - y$ y con un filtro de estimación se estiman los parámetros x que minimizan e. De esta manera las predicciones del modelo $y = h(x)$ se hacen más precisas y por otro lado los parámetros se mantienen con valores actualizados continuamente. Es obvio que el coste computacional de todo este proceso no es despreciable.

Los resultados experimentales para los sistemas basados en tres capas muestran que con estas técnicas se consiguen sistemas elásticos que soportan una buena relación entre el coste de la configuración y la minimización de la violación del SLA. Sin embargo, el cuello de botella que supone la base de datos no se trata en la mayor parte de los trabajos y la escalabilidad horizontal sólo se aplica a la primera y segunda capa.

2.4. Sistemas de Control autoadaptativos basados en Kriging

En las secciones anteriores hemos visto dos técnicas de autoescalado siguiendo la clasificación propuesta por Lorido–Botran et. al. [7]. En esta sección estudiamos una técnica que no tiene una clara catalogación en los diferentes trabajos del estado del arte que hemos analizado en la literatura. Es un modelo basado en el concepto de caja-negra que tiene unas características que lo

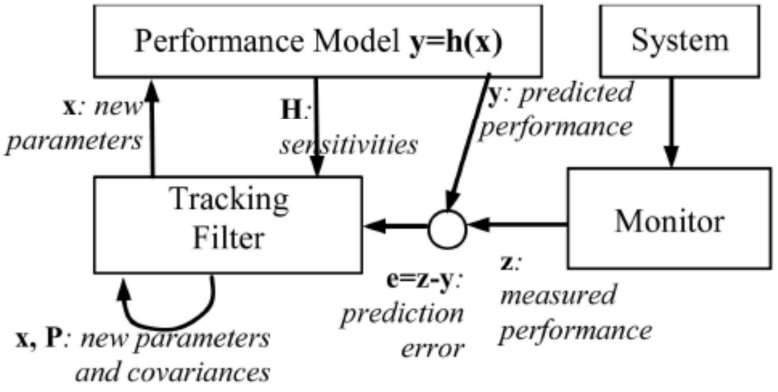

Figura 2.8: Esquema para el seguimiento de la estimación de los parámetros [12]

hacen atractivo para la construcción de controladores de elasticidad.

Gambi propone en [13] sistemas de control autoadaptativos basados en el método de interpolación de Kriging para gestionar la cantidad de recursos necesarios en sistemas cloud y garantizar la calidad del servicio durante su ejecución, a la vez que intenta minimizar los costes económicos. El modelo se aplica en controladores diseñados específicamente para el cloud, y en concreto a nivel de infraestructura, el nivel en el que es posible la creación y destrucción de máquinas virtuales permitiendo la elasticidad del sistema. Así, Gambi, se ha centrado en el bajo nivel y no ha tenido en cuenta los controladores de Kriging a un nivel más alto, como el de plataforma como servicio (PaaS).

La elección de este modelo frente a otros modelos de caja negra o *black-box* ha sido determinada por las siguientes razones:

- Los modelos basados en Kriging son precisos en sistemas que están en ejecución.
- Son modelos robustos frente al ruido y a la imprecisión de los datos de monitorización, además hacen predicciones de forma temporal y pueden ser re-entrenados on-line con poco coste computacional.
- Están basados en una teoría que extiende la regresión tradicional de forma que permite conjuntar medidas de confianza con predicciones.
- Pueden usarse para diseñar controladores proactivos que representan la incertidumbre mientras planifican acciones de control.

2.4.1. Modelos de Caja Negra vs Modelos de Caja Blanca

Según [13] los controladores basados en la construcción de un modelo facilitan el análisis y posteriores actividades de planificación. Hay que tener en cuenta que incluso para cada parámetro

indicado en el SLA se podría definir un modelo diferente. En general cuando se plantea la construcción de un sistema de control, los controladores pueden utilizar modelos de caja blanca o *white-box*, como por ejemplo las redes de colas (*queueing networks*) que hemos visto en las sección anterior, mientras que otros utilizan modelos de caja negra (*black-box*) y modelos de sustitución *surrogate-models*. Los controladores basados en modelos de caja blanca están limitados por el conocimiento de las partes internas del sistema, necesario para construir modelos precisos e imprescindible para obtener predicciones fiables. En los sistemas cloud, no existe esa visibilidad de la parte interna del sistema, y los parámetros de los modelos son difíciles de estimar correctamente, por lo que sólo se pueden obtener descripciones aproximadas del comportamiento del sistema.

Los controladores basados en modelos de caja negra y modelos de sustitución, generan modelos del comportamiento del sistema combinando datos de entrada/salida y técnicas de regresión. Son en general, de más fácil aplicación porque no es necesario conocer los componentes internos del sistema. Además, los modelos de caja negra se pueden readaptar, o volver a entrenar de forma que es posible implementar controladores basados en modelos auto-adaptativos.

Esto son ventajas potenciales porque el modelo de caja negra tiene una penalización y es el tiempo de las pruebas que hay que llevar a cabo para obtener un modelo preciso del comportamiento del sistema. Ese tiempo no es despreciable y añade un sobrecoste a la puesta en marcha del servicio.

2.4.2. Modelos de Kriging

Los modelos de Kriging fueron propuestos inicialmente para aproximar la concentración de minerales valiosos del subsuelo mediante la interpolación de muestras tomadas en diferentes puntos. Desde los años 70 los modelos de Kriging se han usado para desarrollar aproximaciones de funciones de optimización en campos como la geo-estadística, meteorología, aviónica y diseño de circuitos. Más recientemente, los modelos de Kriging se han usado como técnicas de Machine Learning, y son conocidos como Procesos Gausianos. Los modelos de Kriging son una subfamilia de los modelos de caja negra que aproximan funciones no lineales multimodales (las tareas de optimización se centran en encontrar todos o la mayoría de las múltiples soluciones, que al menos sean localmente óptimas, de un problema). Se construyen inicialmente a partir de un conjunto de entrenamiento, y se obtienen las relaciones entre los datos de entrada y salida, un proceso también conocido como entrenamiento supervisado. Una vez entrenados, predicen valores de salida para valores de entrada que no existían en el conjunto de entrenamiento. La figura 2.9 obtenida de [13] muestra como funcionan los modelos de Kriging.

En 2.9 se pueden observar tres secciones. En la sección de la izquierda se presenta el conjunto de entrenamiento, junto con unos inputs (x1, x2) que se desconocen. En la sección central se observa el modelo de Kriging resultante del entrenamiento. Para x1 y x2 se muestra el intervalo de confianza asociado a la predicción. La sección de la derecha muestra el modelo de Kriging completo, donde aparece el área de confianza para todo el espacio de entrada.

Propiedades de los modelos de Kriging:

- Son modelos globales. A diferencia de los modelos de regresión clásicos en los que se promocionan los óptimos locales, los modelos de Kriging abarcan todo el espacio de entrada.
- Cuando los datos no tienen ruido, los modelos de Kriging producen la interpolación exacta, en caso contrario realizan un tipo de *suavizado* de datos.
- Los modelos de Kriging pueden producir una medida de la incertidumbre de sus propias predicciones.
- No necesitan una gran cantidad de datos de entrenamiento para ofrecer predicciones aceptables, por lo que son modelos rápidos de entrenar.

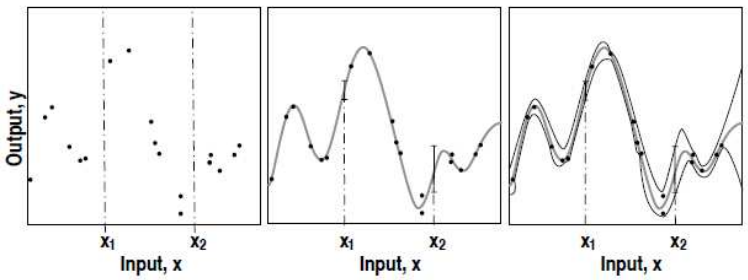

Figura 2.9: Ejemplo de aprendizaje supervisado de un modelo de Kriging [13]

Como se ha visto en la gráfica 2.9 los modelos de Kriging emparejan predicciones con medidas de confianza, y Gambi se sostiene en esta característica para implementar políticas de control robustas que atribuyen pesos a las predicciones del modelo usando sus valores de confianza, para posteriormente tomar decisiones sobre la mejor estrategia de control que implementar. Cuando la confianza de la predicción es baja, los controladores la descartan y pueden aplicar estrategias alternativas. Por ejemplo, los controladores pueden consultar otros modelos, o llevar el sistema a la configuración más cercana y conocida.

Gambi entrena los modelos de Kriging una vez el sistema ya está en ejecución, mediante los datos obtenidos del comportamiento de éste en un periodo de *puesta en marcha* (staging) previo a la puesta en producción. Posteriormente en producción, los controladores cambiarán su comportamiento adaptándose a los cambios en el sistema. Como los controladores obtienen más muestras en producción, la precisión del modelo va creciendo, al igual que el tiempo de generación del modelo. Para evitar que el tiempo de generación del modelo sea excesivamente alto, los controladores filtran muestras antiguas pertenecientes a la misma configuración.

De esta forma, si en el SLA se define un determinado factor, como por ejemplo el tiempo de respuesta medio máximo para cada tipo (o clase) de operación, se contruye mediante Kriging una función

$$avgRT_i = f_i(\#VM_1, ..., \#VM_n, RC_1, ..., RC_m, QL_1, ..., QL_m)$$

donde $avgRT_i$ es el valor del tiempo de respuesta promedio obtenido mediante el modelo para el tipo de operación i_ésima, con i variando de $1...m$; $\#VM_1, ..., \#VM_n$ define la configuración en el número de máquinas virtuales de cada tipo que se disponen en la configuración del sistema entre n tipos distintos; $RC_1, ..., RC_m$ mide la carga del sistema en número de operaciones por segundo recibidas para cada tipo de operación RC_i; y $QL_1, ..., QL_n$ son un grupo de parámetros adicionales que se pueden establecer para cada tipo de operación.

En la siguiente figura 2.10 se muestra el tiempo de respuesta medio calculado para un servidor mediante la función $AvgRT = f(\#AS, RC)$ donde $\#AS$ es el número de MVs en el servidor y RC el número de clientes concurrentes por segundo. La figura también muestra la confianza de la predicción.

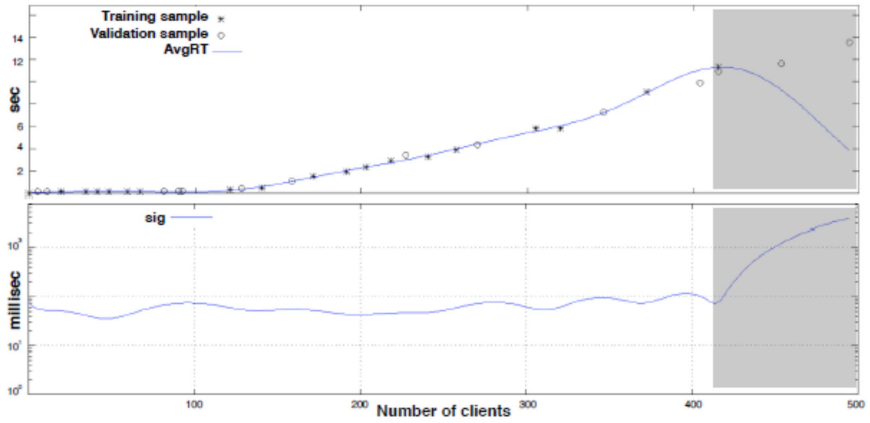

Figura 2.10: Ejemplo de un modelo de Kriging que muestra el tiempo de respuesta medio de un sistema servidor y la confianza de su predicción como función del número de clientes concurrentes por unidad de tiempo para una de las posibles configuraciones del sistema [14]

2.4.3. Arquitectura del Controlador

La figura que se presenta a continuación muestra la arquitectura del controlador propuesto por Gambi 2.11.

El controlador ejecuta dos bucles concurrentes, el bucle de control y el bucle de entrenamiento. El bucle de control obtiene datos de monitorización de las aplicaciones mediante los componentes de tipo *Sensor*. Estos datos de monitorización se envían al modulo de toma de decisiones o *Decision Maker* que implementa la planificación de la configuración. Es este elemento el que llama al *Executor* que se encarga de construir las acciones requeridas, que posteriormente serán enviadas como una secuencia de acciones de control al componente *Actuator* que las ejecuta directamente contra la infraestructura Cloud.

El bucle de entrenamiento implementa la auto-adaptación del modelo. Utiliza información proveniente de los componentes *Sensor* para posteriormente volver a generar el conjunto de entrenamiento en el componente *Model Builder* y entrenar de nuevo el model de Kriging.

El predictor de carga *Workload Predictor* y *SLO monitor* son componentes opcionales también propuestos por Gambi con el objetivo de mejorar el bucle de control de cara al medio y largo plazo, ofreciendo predicciones sobre futuros picos de carga y sobre las veces que se ha sido violado el SLA.

En realidad, el módulo de toma de decisiones es bastante elemental puesto que si, por ejemplo, el tiempo de respuesta medio viola el SLA para una carga de entrada dada, el módulo explora mediante una búsqueda cuál es la configuración que soporta para dicha carga un tiempo de respuesta dentro

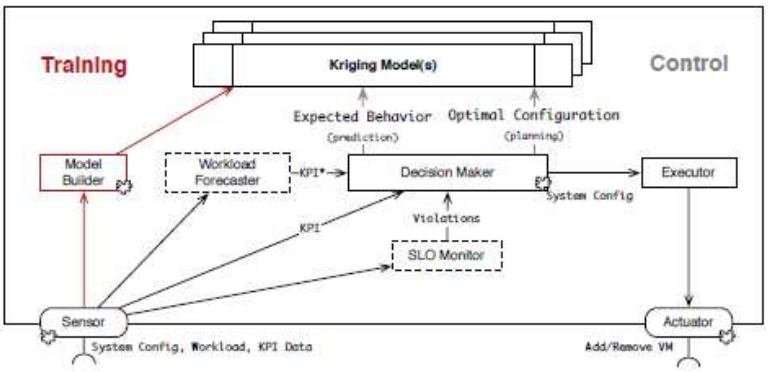

Figura 2.11: Arquitectura del controlador a alto nivel [14]

de los niveles del acuerdo de servicio. Así, establece una clasificación de las posibles configuraciones y para cada configuración calcula su coste económico eligiendo aquella con menor coste como la nueva configuración a instalar en el sistema.

Esta forma de funcionamiento es, a su vez, una limitación puesto que el controlador basado en Kriging puede servir para aquellos sistemas en los que el número de configuraciones a explorar sea razonablemente manejable.

2.4.4. Evaluación Experimental

Las pruebas experimentales llevadas a cabo por Gambi en su tesis doctoral [14] han sido desarrolladas en el IaaS Reservoir Cloud, con el objetivo de experimentar con sistemas reales. Para los casos de estudio ha utilizado las siguientes aplicaciones:

- Sun grid engine: Un sistema de procesamiento batch con arquitectura maestro-esclavo. Los clientes envían trabajo a los maestros, que se encargan de transferirlos a los nodos esclavos que realizan la ejecución y que notifican a los nodos maestros cuándo han terminado. Los nodos esclavos se pueden añadir y eliminar dinámicamente.

- Doodle Web service: Una aplicación web de *2*-capas que implementa un sistema de votaciones. Un balanceador de carga envía las peticiones entrantes al backend del sistema. De vez en cuando los servidores hacen peticiones a la capa de base de datos. En este caso la capa de backend de la aplicación es elástica.

- DoReMap: Un servicio web de tipo restful, que contiene un balanceador de carga que distribuye las peticiones entrantes a un conjunto de motores JOpera, los motores se componen de dos servidores Restful, un servicio web de Doodle y un motor de busqueda llamado RestaurantSearch. También se incluye una capa de base de datos similar a la mencionada en el punto anterior.

Los resultados experimentales indican que el método propuesto para obtener un sistema elástico es preciso y se ajusta a los cambios de carga de forma rápida y adecuada. Las aplicaciones consideradas son un ejemplo sencillo y debería realizarse estudios experimentales con aplicaciones con un mayor número de componentes interactuando entre sí.

2.5. Cuestiones abiertas

A pesar del número importante de trabajos que se han revisado en la literatura no encontramos prácticamente trabajos en los que se aborde claramente la elasticidad sobre despliegues de servicios en PaaS mediante contenedores. De las 34 referencias que se revisan en [7] las aplicaciones desplegadas son del tipo arquitectura de tres capas desplegadas directamente sobre máquinas virtuales. De estas 34 únicamente 6 proponen mecanismos de autoescalado para todas las capas y el resto sólo consideran la elasticidad en la capa de 'business-logic'. Citando el trabajo de Paco Muñoz [15] que es el survey más reciente, queda abierta la cuestión de la 'identificación de la estrategia predictiva y adaptativa que mejor se ajuste para cada tipo de aplicación'.

A este respecto, también hay que indicar que las aplicaciones basadas en microservicios no siguen el modelo tradicional de tres capas por lo que resulta complejo establecer a priori la estrategia de escalado para este tipo de aplicaciones. Como hemos visto anteriormente el único trabajo que hemos encontrado sobre elasticidad con contenedores es bastante simple en cuanto al tipo de servicio que se propone controlar.

Los sistemas basados en reglas son sencillos de entender y de implementar pero basan su funcionamiento en la confianza de las métricas obtenidas y de los niveles máximos programados, por ejemplo, que la carga de la cpu de un contenedor esté por debajo de 0,8. Se ha estudiado experimentalmente en [16] la posibilidad de que las cargas estén por debajo de los umbrales y, sin embargo, el sistema está congestionado aumentando el tiempo de respuesta de las operaciones fuera de los límites admitidos por el SLA. Esto es una motivación para estudiar otros tipos de mecanismos de elasticidad.

En cuanto a las métricas sobre la propia elasticidad de los sistemas no se han seguido las ideas propuestas por Herbst [4] y la práctica experimental no permite establecer de manera adecuada los valores de precisión y velocidad indicados por este mismo autor.

Capítulo 3

Elasticidad mediante Clasificación Binaria

3.1. Introducción

Como se ha visto previamente en este documento, una de las principales ventajas de las Plataformas como Servicio (PaaS) es la facilidad para el despliegue de aplicaciones. Aprovechando esta ventaja, y siendo un PaaS la plataforma donde se ejecutan las aplicaciones basadas en microservicios tiene sentido que sea responabilidad del mismo PaaS el control de la Elasticidad de las aplicaciones.

En el ámbito de la infraestructura (IaaS), los provedores del servicio son los responsables directos de decidir la cantidad de recursos asignados en el sistema al servicio, a parte de mantener y gestionar adecuadamente dicha infraestructura. En el ámbito del PaaS (ideal), los provedores sólo deberían preocuparse por desarrollar los componentes y describir de una forma adecuada cómo están interconectados. El resto, es decir, el despliegue real, la puesta en marcha del servicio y la asignación de recursos al servicio con el objetivo de su buen funcionamiento debería ser responasibilidad de la plataforma como servicio.

Hay que tener en cuenta que el PaaS puede hacer un esfuerzo en garantizar ciertos objetivos del servicio pero no puede hacer escalable un servicio cuyo diseño no ha sido dirigido para tal fin. En el caso de que el servicio haya sido diseñado con las técnicas adecuadas para ser capaz de escalar cuando la carga aumenta, es posible, que el PaaS, de una manera más o menos autónoma, tome las decisiones de escalado oportunas a cada situación de carga sobre el servicio. En un modelo de despliegue del servicio utilizando contenedores, se puede realizar este escalado autónomo mediante la replicación horizontal. Cada componente debe tener asociados un distribuidor de carga y un número de instancias, de forma que, si la carga aumenta se pueden crear nuevas instancias para soportar dicho aumento de carga, y viceversa, si la carga disminuye se pueden destruir instancias.

En este capítulo nos centramos en este último aspecto. Proponemos un diseño de un sistema de control que permite gestionar la escalabilidad horizontal aplicable a los componentes de un servicio desplegado mediante un PaaS. Este controlador implementará la política de elasticidad del PaaS.

El controlador de la elasticidad del PaaS debe resolver los problemas de la asignación de recursos específicos de la aplicación en presencia de dos requisitos anatagonistas: ofrecer la mejor calidad de servicio con el número mínimo de recursos posibles. Cuando hablamos de los recursos que gesiona el PaaS, nos queremos referir al número de instancias (contenedores) que se asignan a un determinado componente del servicio (microservicio). En principio, y por simplicar, podemos asumir que las prestaciones de los contenedores son fijas, aunque en la realidad, los recursos que usa un contenedor podrían variar significativamente. En la mayoría de los casos, estos contenedores

se despliegan sobre máquinas virtuales, con lo que en cierto modo, existe una relación directa entre el número de contenedores y el coste que supone disponer de tal cantidad de recursos.

Aunque no exista un SLA del servicio contratado formalmente con un PaaS, todo servicio debe proveer una Calidad de servicio (QoS) que se ve mejorada en función del aumento de las prestaciones y de la cantidad de recursos asignados al mismo. Por otra parte, minimizar los costes de operación se obtiene decrementando la cantidad de recursos asignados al servicio. Este hecho marca el objetivo de la elasticidad, básicamente: gestionar los recursos asigandos para mantener la capacidad de escalado de la aplicación evitando sobre-asignaciones innecesarias en presencia de cargas de trabajo que fluctuan en el tiempo.

A lo largo de los capítulos anteriores se han repasado técnicas conretas para el diseño de los controladores de la elasticidad pero, como ya indicamos, estas técnicas de autoescalado estaban centradas en la asignación de máquinas virtuales, cuyo número es en pincipio bastante menor que el número de instancias de componentes de un servicio. De hecho, si repasamos la técnica de Kriging, el número de componentes del servicio eran tres y el número de MVs no superaba una docena. En el caso de los servicios actuales, y en concreto de microservicios, el número de componentes del servicio puede llegar a ser mucho mayor, puesto que la tendencia actual, es desarrollar un servicio como composición de un gran número de unidades (microservicios), en la que cada uno de ellos es responsbale de una funcionalidad concreta. Esto supone un nuevo reto en el diseño de los controladores puesto que cambia la forma de escalar los servicios, además los problemas de explosión de información debido al incremento de componentes pueden aparecer de manera natural.

En el estado del arte, hemos visto que la mayor parte de los sistemas reales ofrecen mecanismos de elasticidad basados en reglas. Estos mecanismos se basan en asumir un comportamiento lineal y estable entre los recursos y los criterios de calidad del servicio. En la práctica, el sistema se comporta de forma mas compleja y es bien conocido, que los problemas de contención del hardware y el software invalidan estas suposiciones. Por otro lado, las medidas que se toman sobre los componentes se ven afectadas de forma impredecible por diversos factores, muchas veces desconocidos al diseñador. Esto ha supuesto ofrecer diversas alternativas a los sistemas basados en reglas mediante otras aproximaciones, como la teoría de control o los modelos de colas. En ese sentido, la idea de utilizar controladores adaptativos parece la mejor aproximación para el diseño de un control de la elasticidad. La cuestión es cómo diseñar un controlador adaptativo sin disponer de un conocimiento a priori completo del funcionamiento del servicio y de las necesidades de recursos que en cada caso necesita.

Las principales ideas para el diseño de este tipo de controladores adaptativos han sido las siguientes:

- A partir de una descripción del grafo de componentes (que define los componentes del servicio y cómo se encuentran interconectados), Zheng [12] propone definir un modelo sencillo de colas que describe el funcionamiento del servicio. En el modelo, los parámetros que definen su funcionamiento son desconocidos. En la fase de puesta en marcha del servicio, aplica un sistema de estimación de parámetros para, a partir de las medidas reales del sistema (carga de entrada, tiempos de respuesta, utilizaciones de los dispositivos), estimar los parámtros del modelo anáitico. El sistema de estimación puede estar ejecutándose de forma continua y modificando los parámetros del modelo en cada momento. Como en el caso de un sistema de control adaptativo, dispone, por tanto, de un subsitema de estimación del modelo del sistema bajo control. En el caso de producirse alguna violación del SLA (referido a un tiempo máximo de respuesta) utiliza el modelo estimado para buscar una nueva configuración del servicio que se ajuste a los nuevos requisitos de la carga de entrada. Esta búsqueda es un proceso que se puede realizar mediante técnicas conocidas como el Hill Climbing o empleando otras más sofisticadas como el descenso por gradiente. En todo caso, puede almacenar información

histórica sobre los modelos obtenidos, cargas soportadas y tiempos de respuesta para, en base a esta información, encontrar la configuración que se ajuste mejor a la carga para satisfacer el tiempo de respuesta del SLA.

- El modelo que utiliza Gambi [13][14] consiste simplemente en emplear una función que devuelve una estimación del tiempo de respuesta, en relación a los siguientes parámetros: la configuración del servicio (número de MVs asignadas a cada componente en un modelo de tres capas), la carga de entrada (puede ser por cada tipo de operación al servicio), y una estimación del tamaño de las peticiones en espera. Esta función se aprende en la fase de establecimiento del servicio, y puede reajustarse cuando el servicio ya se encuentra operativo. La idea de Gambi no consiste en realizar todas las posibles pruebas (esto sería inviable) para obtener la mejor función posible, sino en aprovecharse de las capacidades de interpolación de los métodos de Kriging. En el caso de producirse una violación del tiempo de servicio definido en el SLA, el controlador simplemente consiste en generar todas las posibles configuraciones (es un número no muy grande) y encontrar directamente la configuración con el menor número de recursos (MVs) que ofrezca el tiempo de respuesta adecuado para el cumplimento del SLA.

- El método que utiliza Jamshidi [20] se basa en definir el estado de todo el servicio a partir de la pareja de valores formados por la carga de entrada y el tiempo de respuesta, y utilizar un controlador basado en técnicas fuzzy para establecer la acción de control que sea necesaria para evitar que el tiempo de respuesta viole el SLA. Las acciones de control se aprenden mediante un proceso de aprendizaje denominado Q-learning. En la fase de operación del servicio, el algoritmo de Q-learning elige de forma aleatoria la acción de control para el estado actual y una vez aplicada, se analiza una función de utilidad para el nuevo estado obtenido. El proceso continúa a medida que el servicio está en funcionamiento y el algoritmo converge hasta encontrar las acciones adecuadas que minimizan la función de utilidad. El método de Q-learning es una variación del método de aprendizaje reforzado que se utiliza en los modelos de Markov. La función de utilidad depende de la carga, del número de MVs utilizadas y del tiempo de respuesta obtenido.

- Liu en [19] utiliza un esquema similar al anterior pero sin un controlador fuzzy. Emplea una variación del aprendizaje reforzado en el que primero utiliza una estrategia agresiva. Cuando se viola el SLA introduce un número 'suficiente' de MVs que garanticen que se va a satisfacer el SLA y a partir de ese punto, va ajustando el aprendizaje reforzado.

Los métodos anteriores tienen un coste computacional muy alto cuando el número de componentes a controlar es significativo. Los métodos de aprendizaje reforzado se utilizan para un sólo componente (un servidor que puede estar instalado en un MV) y controla el número de MVs que tienen que instalarse para soportar la carga de entrada. El tiempo de convergencia de los algoritmos es lento hasta llegar a ajustarse. Los métodos que utilizan un modelo 'sustitutivo' del servicio se emplean normalmente para, como mucho, un sistema de tres capas (tres componentes ver figura 2.6), donde cada capa se instala en un número de MVs.

La idea base de nuestro trabajo consiste en disponer de un modelo adaptativo que permita clasificar las configuraciones en función de su comportamiento frente a la carga de entrada para conocer si el tiempo de respuesta de la configuración para dicha carga satisface o no el SLA definido. El controlador se basa en la estimación del clasificador de manera que se pueda elegir una configuración con el menor número de instancias que satisfaga el SLA. En este trabajo nos basamos en los sistemas de clasificacion binaria que aprenden continuamente ('online') en concreto hemos utilizado el algoritmo de clasificación denomiado AROW [18]. Este algoritmo de clasificación tiene varias características interesantes:

- La clasificación tiene un coste computacional bajo, simplemente realiza un producto escalar de dos vectores.
- Su aprendizaje se basa en el margen del producto y en una matriz de varianza, y su coste computacional es cuadrático en la dimensión del vector.
- En otras aplicaciones se ha demostrado capaz de tratar el ruido de clasificación y ofrece un alto porcentaje de acierto.

El resto del capítulo se organiza de la siguiente forma. En la sección 3.2 se explica el modelo de componentes de un servicio mediante un ejemplo y cómo podemos simular su comportamiento usando la teoría de colas. En la sección 3.3, se exponen las ideas prinicpales en las que se basa el controlador de elasticidad. Posteriormente, en la sección 3.4 presentamos los fundamentos de la clasificación binaria y el algoritmo AROW [18] utilizado. En la sección 3.5 se detalla el controlador de elasticidad que hemos propuesto. En la sección 3.6 presentamos una variación del algoritmo de búsqueda Hill-climbing que empleamos para encontrar la mejor configuración cuando se produce una violación del SLA. Finalmente, en la sección 3.7 presentamos los resultados experimentales obtenidos con el controlador de elasticidad propuesto. Las conclusiones se presentarán en el capitulo final de esta memoria.

3.2. Modelo de los componentes de un servicio

Un modelo basado en microservicios como el que se observa en la figura 3.1 (obtenido del curso de introducción a los microservicios del blog de Nginx [6]), consiste en dividir una aplicación en áreas funcionales concretas para posteriormente implementarlas de forma independiente. Al dividir un servicio en distintas partes independientes, se consigue que sea más sencillo replicar el microservicio, sólo haría falta un balanceador de carga entre el cliente y el microservicio. No obstante no todo son ventajas, puesto que el desarrollo de este tipo de aplicaciones requiere una exhaustiva definición de las interfaces de comunicación. A menudo para llevar a cabo esta comunicación se utilizan APIs REST o paso de mensajes de tipo publicador/suscriptor.

En la figura 3.1 podemos encontrar ocho microservicios, que en conjunto ofrecen la funcionalidad completa de una aplicación para contratar un servicio de transporte. Se puede observar como cada uno tiene una interfaz tanto para comunicarse con los usuarios (GUI) o con el resto de microservicios (REST API). El componente API gateway actua de balanceador y director de las peticiones que no se hacen desde una GUI sino directamente.

En un caso de ejemplo, imaginemos que hay un evento importante, y la empresa que ha desarrollado esta aplicación es la encargada de trasladar a los asistentes. Al terminar el evento todas esas personas cogen su dispositivo móvil para solicitar un transporte. En ese momento el componente encargado de servir los conductores disponibles se satura y deja de dar un servicio que cumple un cierto tiempo de respuesta según un SLA. Al ser una arquitectura no monolítica es muy sencillo duplicar el microservicio, es decir, añadir una instancia de ese componente, y poner delante algún tipo de balanceador de carga, como un proxy o un DNS rotativo [1].

De esta manera, cada componente principal del servicio puede estar formado por una o más instancias. Esto permite que el servicio escale de manera horizontal por medio de la duplicación de las instancias. Cada componente tiene asociado un balanceador de carga, de forma que éste puede repartir la carga entre las distintas instancias con objeto de mejorar las prestaciones del componente.

[1] A menudo este tipo de soluciones también llamadas Service Discovery se incluyen en los PaaS para facilitar el funcionamiento de las aplicaciones en este tipo de casos.

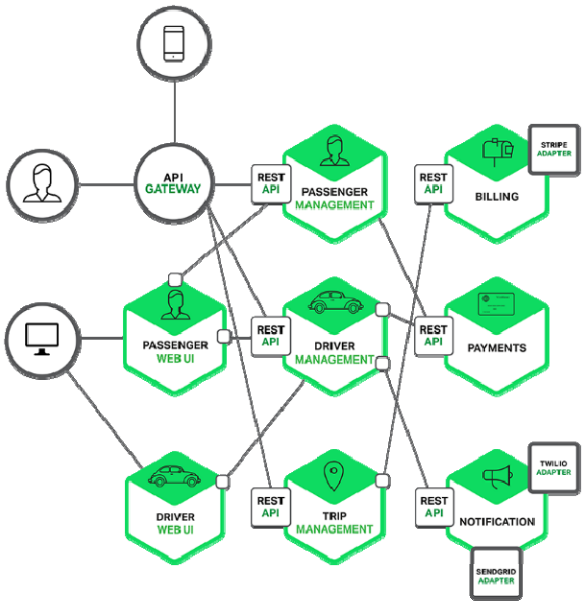

Figura 3.1: Componentes de un servicio basado en una arquitectura de micro-servicios

Mediante un modelo análitico, basado en la teoría de colas, podemos simular el comportamiento de un componente principal y sus instancias como aparece en la figura 3.2.

Un componente con n instancias, donde cada instancia recibe una parte proporcional de la carga de entrada al componente, puede aproximarse mediante un sistema multiservidor con un cierto retardo (que depende del número de instancias) con el objeto de simular un cierto grado de congestión provocado por el consumo y gestión de recursos a medida que aumenta el número de instancias. Esta aproximación es la aproximación propuesta por Seidmaan [21].

Nuestro objetivo no es modelar de forma muy precisa el comportamiento de un servicio. Para el propósito de analizar y diseñar un posible sistema de elasticidad nos es suficiente con considerar un sistema aproximado. El modelo se considera un modelo abierto estacionario, es decir, el sistema puede recibir un número no acotado de peticiones, y la carga de entrada es igual a la productividad obtenida. También suponemos que el sistema es 'single-class', es decir, sólo consideramos un único tipo de operación. Finalmente, la carga λ (número de operaciones/unidad de tiempo), se supone que sigue una distribución exponencial. Este hecho permite seguir el modelo de Jackson en forma de producto. El algoritmo para el cálculo del tiempo de respuesta en función de la carga es bastante simple porque el modelo de Jackon establece que, en las condiciones anteriores, no importa lo complejo que sea el grafo de interconexión de los componentes, el tiempo de respuesta es la suma de los tiempos de respuesta de una cadena formada por todos los componentes (como si fuesen unos detrás de otros).

Siguiendo lo anterior, supongamos que disponemos de un sistema con K componentes principales. Cada componente k-ésimo, en un determinado momento, puede tener N_k instancias. El vector de configuración del servicio es por tanto $\overrightarrow{N}$ de dimensión K. Una operación sobre el servicio de-

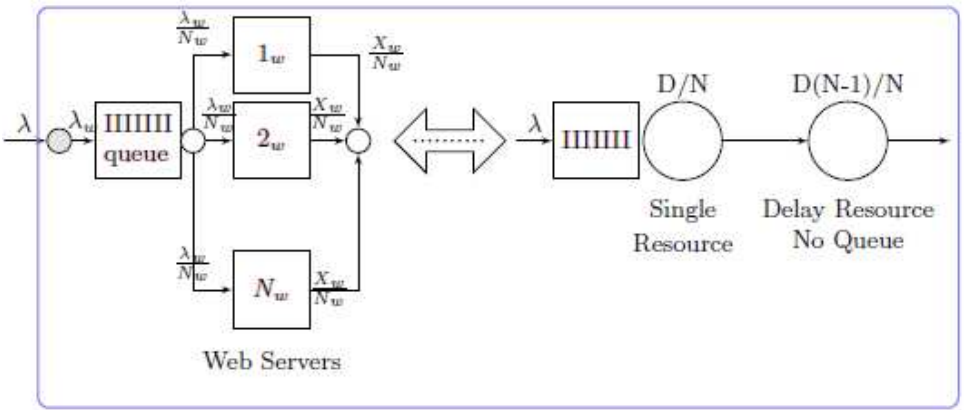

Figura 3.2: Aproximación de Seidmann

manda de cada componente k-ésimo un cierto tiempo D_k (recordemos que la demanda refleja el número de visistas, que una operación necesita de un componente, multiplicado por el tiempo de servicio de dicho componente). Con todo ello, el tiempo de respuesta medio para una carga de entrada de λ se calcula como sigue (ver Apéndice A):

$$T = \sum_{i=1}^{K} T_k \tag{3.1}$$

$$T_k = D_k \frac{N_k - 1}{N_k} + \frac{D_k}{1 - \lambda D_k / N_k} \tag{3.2}$$

El valor $\lambda D_k / N_k$ representa el factor de utilización del componente k-ésimo con N_k instancias. Si este valor es próximo a la unidad el tiempo de respuesta tiende a infinito. Por otro lado, la presencia del término $D_k \frac{N_k - 1}{N_k}$ de la aproximación de Seidmann, evita que el sistema tenga una escalabilidad arbitraria (aumentando de manera indiscriminada el número de instancias).

De esta manera, el modelo del sistema propuesto lo vamos a utilizar para las pruebas del controlador de elasticidad como si fuese un sistema real. El modelo abierto, por sus características, es adecuado para el estudio experimental ya que a medida que la carga de entrada aumenta tiende a la saturación de los componentes y el tiempo de respuesta aumenta rápidamente en cuanto un componente comienza a tener un factor de utilización por encima de 0,9. Para las pruebas vamos a emplear un sistema con $K = 6$ componentes principales. Cada componente puede tener un máximo de 10 instancias. Las demandas forman un vector con valores $[0{,}01, 0{,}03, 0{,}05, 0{,}012, 0{,}065, 0{,}04]$. En este caso podemos representar la curva de tiempo de respuesta máximo frente a la carga máxima que

puede soportar una configuración con x instancias por componente (donde x varía de 1 a 10), cuando alguno de sus componentes tiene un factor de utilización de u, con u variando de 0,7 hasta 0,9. Como observamos en la figura 3.3, el sistema puede soportar una carga máxima de aproximadamente 140 operaciones por unidad de tiempo[2] generando un tiempo de respuesta de aproximadamente 1,2 cuando la configuración tiene el máximo número de instancias por componente, 10 instancias, y el factor de utilización de algún componente alcanza 0,9.

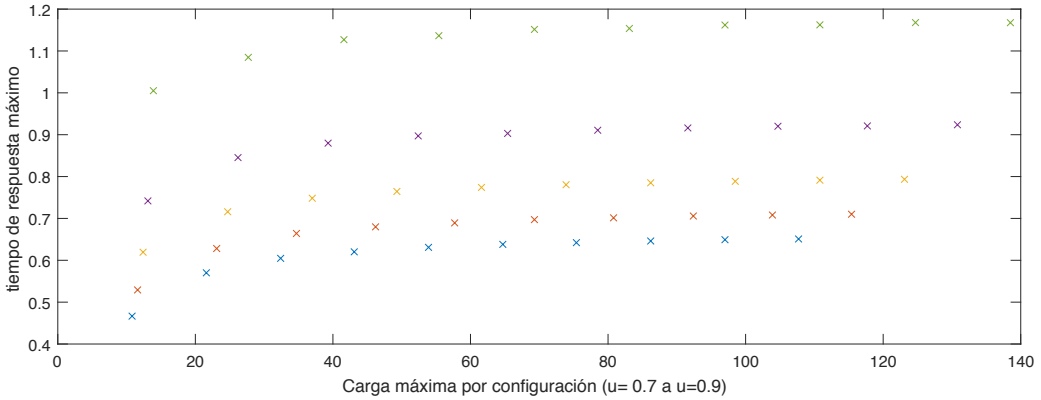

Figura 3.3: Tiempos de respuesta frente a la carga máxima para diferentes configuraciones

3.3. Control de Elasticidad sin conocimiento previo del modelo

En la fase de establecimiento del servicio se deben definir los parámetros y valores que forman parte del SLA. Normalmente se indican la disponibilidad del sistema, así como el tiempo de respuesta máximo para las diferentes operaciones. En este trabajo nos centramos en el tiempo de respuesta máximo. En el ejemplo anterior podemos establecer como valor de tiempo de respuesta máximo 1,0. Este valor lo denominamos genéricamente SLA_UP, SLA_UP=1,0. El servicio puede comenzar en una determinada configuración $\overrightarrow{N}(t_0)$ y soportar una carga que no viole el SLA_UP. Si el tiempo de respuesta es mayor que el SLA_UP el control de elasticidad debe reaccionar para generar una nueva configuración $\overrightarrow{N}(t)$ que permita mantener el tiempo de respuesta por debajo de SLA_UP. El controlador de elasticidad que proponemos en este trabajo es en pricipio reactivo, es decir, no realiza ninguna predicción de la carga para adelantarse a los acontecimientos. Hay que tener en cuenta que una vez que se genera la nueva configuración $\overrightarrow{N}(t)$ esta debe ser instalada, y eso supone un cierto retardo durante el cual, el tiempo de respuesta puede seguir violando el SLA_UP.

El controlador de elasticidad que proponemos no tiene ningún conocimiento del modelo del servicio y sólo recibe como información la configuración actual $\overrightarrow{N}(t)$, la carga $\lambda(t)$, y el tiempo de respuesta observado $TR(t)$. Bajo estas consideraciones el controlador sólo puede tomar una decisión cuando se viola el SLA_UP y es añadir instancias. El número de instancias en principio no es conocido pero se puede seguir la idea de Liu [19] y añadir de una forma agresiva un número suficiente de instancias que restablezca el valor del tiempo de respuesta. Esta estrategia no es óptima y puede producir un sobredimensionado de la configuración generando un tiempo de respuesta innecesariamente bajo. Aunque en el SLA no se define un límite inferior para el tiempo de respuesta,

[2]En el trabajo experimental no se establece la unidad de tiempo.

por cuestiones de optimización podemos proponer un tiempo SLA_DOWN[3] de manera que si el tiempo de respuesta está por debajo de este valor, el controlador de elasticidad quita instancias (en principio, de manera aleatoria). Esta estrategia que hemos indicado es una estrategia 'Agresiva-relajada'.

A medida que el sistema está funcionando la información $\overrightarrow{N}(t)$, $\lambda(t)$, SLA_UP, $up(t) \in \{-1, +1\}$, SLA_DOWN(t), y $down(t) \in \{-1, +1\}$, puede utilizarse para aprender de manera 'on-line' un clasificador binario que sea la base para clasificar las configuraciones y realizar de manera más óptima la elección de la configuración que mejor se ajuste a la relación carga tiempo de respuesta. De esta forma, podemos combinar diferentes estrategias bajo el mismo control de elasticidad, además el control es adaptativo puesto que el clasificador binario puede adaptar su aprendizaje de forma 'on-line'. En la siguiente sección indicamos las principales características del clasificador binario que vamos a utilizar en nuestro trabajo.

3.4. Técnicas de clasificación binaria de aprendizaje continuo

En esta sección introducimos los conceptos fundamentales en los que se basan los sistemas de clasificación binaria de aprendizaje continuo. La clasificación binaria continua tiene lugar en una secuencia de periodos o rondas. En cada ronda, el algoritmo de clasificación observa los valores de la muestra de entrada y predice su etiqueta de clasificación entre dos únicos valores $+1$ o -1. Una vez realizada la predicción, el valor real de la etiqueta a la que pertenece la muestra de entrada es presentada al algoritmo de aprendizaje del clasificador que posiblemente sufra una pérdida instantanea, la cual refleja el grado de equivocación de la predicción. Al final de cada ronda, el algoritmo de aprendizaje utiliza el par (muetra, etiqueta real) para mejorar la capacidad de predicción del clasificador. Cada muestra de entrada presentada al clasificador en la ronda t, es un vector n dimensional $\overrightarrow{x}(t) \in \mathcal{R}^n$. Asumimos que $\overrightarrow{x}(t)$ tiene asociada una única etiqueta $y(t) \in \{+1, -1\}$. Cada pareja $(\overrightarrow{x}(t), y(t))$ es un ejemplo para el algoritmo de aprendizaje del clasificador. El clasificador binario utiliza una función de clasificación que se mantiene en memoria interna y que el algoritmo de aprendizaje actualiza de ronda en ronda. En este trabajo vamos a emplear funciones de clasificación basadas en un vector de pesos $\overrightarrow{\mu} \in \mathcal{R}^n$, de forma que la función de clasificación que realiza la predicción es muy simple:

$$\texttt{sign}(\overrightarrow{\mu} \cdot \overrightarrow{x}(t)) = \widehat{y}(t) \tag{3.3}$$

donde $\cdot$ representa la operación producto escalar de vectores, y el valor $\widehat{y}(t)$ es el resultado de la predicción del clasificador para la muestra dada.

La magnitud $|\overrightarrow{\mu} \cdot \overrightarrow{x}(t)|$ se puede interpretar como un grado de confianza de la predicción. La tearea del algoritmo de aprendizaje asociado al clasificador consiste en aprender de forma incremental el vector de pesos $\overrightarrow{\mu}$. Bajo este prisma, se denota por $\overrightarrow{\mu}(t)$ el vector de pesos que se aplica en la ronda t-ésima. El margen con signo es el término $y(t) \times (\overrightarrow{\mu}(t) \cdot \overrightarrow{x}(t))$. Si este margen es positivo entonces $\texttt{sign}(\overrightarrow{\mu} \cdot \overrightarrow{x}(t)) = \widehat{y}(t) = y(t)$, es decir, el clasificador realiza una clasificación correcta. No obstante, el hecho de que la clasificación sea correcta no indica un margen adecuado, por ese motivo es conveniente que la predicción tenga un valor alto de confianza. El objetivo de un algoritmo de aprendizaje es garantizar un margen con valor más grande que 1. Si esto no es así, el

[3]Ese límite inferior se deriva de la solución a un problema de optimización de costes, pero no guarda relación directa con el tiempo de respuesta, ni con ningún compromiso entre usuario y proveedor. Es una optimización interna que debe realizar el proveedor.

clasificador produce una pérdida instantanea definida por la función *hinge-loss* definida como:

$$\ell(\overrightarrow{\mu};(\overrightarrow{x},y)) = \begin{cases} 0 & y \times (\overrightarrow{\mu} \cdot \overrightarrow{x}) \geq 1 \\ 1 - y \times (\overrightarrow{\mu} \cdot \overrightarrow{x}) & \text{en otro caso} \end{cases} \tag{3.4}$$

Se observa que si el margen excede el valor 1, la pérdida es cero. La cuestión para el algoritmo de aprendizaje del clasificador es simplemente minimizar la función $\ell(t) = \ell(\overrightarrow{\mu}(t);(\overrightarrow{x}(t),y))$ en el intervalo de rondas considerado T. Otra forma de ver el problema del aprendizaje es acotar la acumulación de las pérdidas cuadráticas $\sum_{t=1}^{T} \ell(t)^2$ y por tanto, limitar el número de errores de predicción realizados en el intervalo considerado.

K. Crammer et al. [17] estudian diferentes algoritmos de aprendizaje del vector de pesos para la clasificación binaria. En este trabajo utilizaremos una versión mejorada de los algoritmos en [17] propuesta por Crammer et al. en [18]. El algoritmo de aprendizaje se denomina 'Arow' (Adaptive regularization of weights). El algoritmo de aprendizaje ajusta $\overrightarrow{\mu}(t)$ a medida que aparecen nuevas muestras pero también ajusta una matriz de covarianza con el objeto de capturar la interacción entre pesos. El algoritmo es el siguiente [18]:

Parámetro de entrada: r
Inicialización: $\overrightarrow{\mu}(t_0) = \overrightarrow{0}$, $\Sigma(t_0) = I$ (Σ matriz de covarianza $n \times n$).
En la ronda t-ésima $t = 1...T$

- Recibe la muestra $\overrightarrow{x}(t) \in \mathcal{R}^n$
- Calcula el margen y confianza $m_t = \overrightarrow{\mu}(t-1) \cdot \overrightarrow{x}(t)$; $v_t = (\overrightarrow{x}(t))^T \Sigma(t-1) \overrightarrow{x}(t)$
- Recibe la etiqueta real $y(t)$, y produce una pérdida $\ell(t) = 1$ si $\texttt{sign}(m_t) \neq y(t)$
- Si $y(t)m_t < 1$, utiliza las siguientes ecuaciones para actualizar el vector de pesos y la matriz de covarianza:
  $\beta_t = \dfrac{1}{v_t + r}$
  $\alpha_t = \texttt{max}(0, 1 - y(t)m_t) \times \beta_t$
  $\overrightarrow{\mu}(t) = \overrightarrow{\mu}(t-1) + \alpha_t \Sigma(t-1) y(t) \overrightarrow{x}(t)$
  $\Sigma(t) = \Sigma(t-1) - \beta_t \Sigma(t-1) \overrightarrow{x}(t) (\overrightarrow{x}(t))^T \Sigma(t-1)$
- **salida**: vector de pesos $\overrightarrow{\mu}(t)$, confianza $\Sigma(t)$

El coste computacional del clasificador es lineal puesto que es simplemente un producto escalar de dos vectores. En cuanto al coste computacional del algoritmo de aprendizaje del calsificador es cuadrático puesto que require el uso del producto de la matriz de covarianza por un vector. Este clasificador ha sido empleado en diversas aplicaciones mostrando buenos resultados en cuanto al porcentaje de aciertos obtenidos. Para más detalles se puede consultar la referencia [18].

3.5. Algoritmo de control

El controlador propuesto en este trabajo (algoritmo 1) tiene como parámetros de entrada los requisitos descritos en el SLA, en concreto el límite superior del tiempo de respuesta a partir del cual se violaría el SLA (SLA_UP), y el límite inferior (SLA_DOWN), el cuál indica que si el tiempo de respuesta es inferior el coste sería demasiado alto, por lo que no merece la pena que la aplicación necesite más recursos para estar por debajo de ese límite, así estos dos parámetros establecen una franja ideal para el tiempo de respuesta de la aplicación. Para que la aplicación se mantenga en esta franja nuestro controlador utiliza el clasificador binario AROW, en concreto utiliza dos instancias

de éste, una para clasificar el tiempo de respuesta en base a si supera o no el límite superior del SLA y otra para clasificar si el tiempo de respuesta es inferior al límite inferior del SLA. Cada instancia del AROW, viene dada por dos parámetros, el U un vector de pesos, y E una matriz de covarianzas. Internamente, el algoritmo del AROW es exactamente el mismo.

En primer lugar, es necesario inicializar estos parámetros del AROW, junto con la configuración inicial o de despliegue del servicio. Mientras el servicio esté ejecutándose se presupone que cada cierto tiempo se recibirán las métricas de carga y de tiempo de respuesta de la aplicación. Una de las ventajas del controlador propuesto es que no necesita complicados sistemas para obtener una gran cantidad de métricas, y el tiempo de respuesta y la carga son fáciles de obtener en un sistema real. Como se detalla más adelante, en las pruebas experimentales, para generar la carga se han utilizado distintos simuladores, mientras que a partir de esta carga se ha utilizado el Modelo Abierto basado en la teoría de colas para generar los tiempos de respuesta.

Una vez obtenidas estas dos métricas, se establecen los valores para las variables *up* y *down* que se encargan de guardar si el tiempo de respuesta a superado el SLA (*1*) o no (*-1*) tanto por arriba en el caso de *up* como por abajo en el caso de *down*. Posteriormente, se realiza la misma operación, pero consultando la predicción realizada por los AROW, estableciendo así los valores de las variables *ArowUp* y *ArowDown*. Una vez hecho se comprueba por separado si la predicción de cada AROW ha fallado, para el AROW que haya fallado se llama al algoritmo de clasificación del AROW para que aprenda, de forma que para la configuración actual AROW sabrá que viola el tiempo de respuesta establecido.

Después de la fase de clasificación de los AROWs se encuentra la fase de toma de decisiones del controlador, en la que, según los valores de las variables explicadas anteriormente, se tomará una decisión, aplicando diferentes estrategias para cada caso y para cada versión del controlador.

Así hemos decidido llevar a cabo tres versiones diferentes del controlador dependiendo de la estrategia utilizada para encontrar la mejor nueva configuración:

- Estrategia Agresiva/Relajada: En caso de que el tiempo de respuesta esté por debajo del tiempo de respuesta up definido, utiliza una estrategia agresiva añadiendo una instancia a cad uno de los compentes de la aplicación, en caso de que haya que reducir el número de instancias porque el tiempo de respuesta sea más rápido que el límite inferior definido se utiliza una estrategia más conservadora, eliminando una instancia a uno o dos componentes elegidos de forma aleatoria. Con esto se consigue que la aplicación reaccione más rapidamente a las subidas de carga y que, posteriormente, poco a poco vaya eliminando instancias hasta encontrarse en la franja idónea definida.

- Estrategia de búsqueda mediante HillClimbing modificado: Para esta versión se utiliza un algoritmo de tipo HillClimbing modificado que se encarga de encontrar la mejor configuración a partir de los vecinos de la configuración actual. Para las estrategias de de *ArowUp* (*ArowUpFailsStrategy* y *ArowUpHitsStrategy* en el algoritmo) se utiliza una modificacin pensada para que los vecinos tengan un mayor número de instancias, en cambio para las estrategias de *ArowDown* (*ArowDownFailsStrategy* y *ArowDownHitsStrategy* en el algoritmo) se utiliza una modificación en la que los vecinos tengan un número menor de instancias.

- Estrategia Combinada: Combina las dos versiones que acabamos de mencionar, utilizando el Hill Climbing en caso de que el AROW haya acertado, de esta forma se consigue una respuesta intermedia frente a las dos versiones independientes a la hora de alcanzar la franja ideal del tiempo de respuesta.

```
Program ElasticityController
    input : Service configuration config, SLA upper bound SLA_UP, SLA lower bound
            SLA_DOWN

    // Var initialization
    initialize arowUpU;
    initialize arowUpE;
    initialize ArowDownU;
    initialize ArowDownE;

    initialize config;

    while service is working do
        load ← GetLoad();
        responseTime ← GetResponseTime();
        if responseTime > SLA_UP then up ← 1;
        else up ← -1;
        if responseTime ≥ SLA_DOWN then down ← -1;
        else down ← 1;

        if Dot(arowUpU, [config, load]) > 0 then arowUp ← 1;
        else arowUp ← -1;
        if Dot(ArowDownU, [config, load]) > 0 then ArowDown ← 1;
        else ArowDown ← -1;

        // AROW learning when fails
        if arowUp ≠ up then [arowUpU, arowUpE] ←Arow(arowUpU, arowUpE, [config, load],
         up);
        if ArowDown ≠ down then [ArowDownU, ArowDownE] ←Arow(ArowDownU,
         ArowDownE, [config, load], down);

        // Decission making
        if up = 1 and arowUp = −1 then
            // arowUp fails
            config ← ArowUpFailsStrategy();
        end
        else if up = 1 and arowUp = 1 then
            // arowUp hits
            config ← ArowUpHitsStrategy();
        end

        if down = 1 and ArowDown = −1 then
            // ArowDown fails
            config ← ArowDownFailsStrategy();
        end
        else if down = 1 and ArowDown = 1 then
            // ArowDown hits
            config ← ArowDownHitsStrategy();
        end

        ApplyConfig (config);
    end
end
```

Algorithm 1: Controlador de Elasticidad

3.6. Búsqueda de la mejor configuración

Como se ha comentado en la sección anterior, hemos decidio utilizar para una de las versiones del controlador un Hill Climbing modificado que nos permita encontrar la mejor configuración a partir de los vecinos de la configuración actual. Para ello hemos tomado como base el algoritmo genérico de Hill Climbing, y lo hemos modificado como se puede ver en el algoritmo 2. En primer lugar se obtienen los vecinos de la configuración actual junto con ciertos datos que son importantes para el resto del algoritmo. Las configuraciones vecinas en el caso de utilizar el HillClimbing Modificado se van a obtener mediante la suma de una instancia a cada componente de la configuaración, así si el servicio tiene seis componentes el número de vecinos será también seis, donde la suma del total de instancias de cada vecino es la suma del total de instancias de la configuación incial más uno. Cada vecino necesita tener cierta información asociada para poder llevar a cabo la búsqueda de la mejor configuración. Esa información incluye los valores de la predicción de *ArowUp* y *ArowDown* para cada vecino, tanto el signo como el valor de confianza, valores obtenidos a partir del producto escalar del vecino y la carga con el vector de pesos de los *Arow*. Además, cada vecino debe incluir un bit, para indicar si la configuración es elegible o no. Las características para que un vecino no sea elegible son las siguientes:

- El número de instancias del vecino para un componentes se pasa un valor establecido (a dos en las pruebas experimentales) de la configuracón actual.

Si todos los vecinos de la configuración actual tienen ese bit a cero, significa que no hay ninguno elegible, por lo que se devuelve la misma configuración, tal como se puede ver en 2 Posteriormente en caso de que exista algún vecino elegible como nuevo candidato se comprueba si el estado del ArowUP, en caso de que todos los vecinos tengan valor ArowUp a uno, significa que el Arow no ha predecido uno que satisfaga totalmente la subida de instancias necesaria, no obstante, se devuelve el valor qué tiene el valor de confianza más cercano a cero, es decir, el menos malo de todas las opciones, en caso de que exista alg]'un elemento con valor de *ArowUp* a menos uno, quiere decir que es idoneo, y más aún si su valor de *ArowUp* es menos uno. Si no existe ningun elemento con valor de *ArowUp* y *ArowDown* a menos uno, el algoritmo devuelve de nuevo el vecino que tiene el valor de confianza más cercano a cero de entre todos aquellos que tenían valor *ArowUp* a menos uno.

```
Function HillClimbingForArowUp
    input  : Service configuration actualConfig, Load and Arow args auxArgs
    output: Best found neighbour

    neighbours ← GetNeigbours(actualConfig, auxArgs);
    if exists some appropiate neighbour in neighbours then
        if all candidates from neighbours satisfy arowUp = 1 then
            neighbour ← neighbour with arowUp value nearest to 0 from neighbours;
            return HillClimbingForArowUp(neighbour, auxArgs);
        end
        else
            neighboursFiltered ← filter neighbours with arowUp = −1;
            if all candidates from neighboursFiltered satisfy ArowDown = 1 then
                return neighbour with ArowDown value nearest to 0 from neighboursFiltered;
            end
            else
                return random neighbour with ArowDown = −1 from neighboursFiltered;
            end
        end
    end
    else
        return actualConfig;
    end
end
```

Algorithm 2: Hill Climbing Modificado para usarlo en una estrategia de aumento de instancias

3.7. Evaluación experimental

A continuación se presentan las pruebas experimentales realizadas para comprobar la valided del controlador propuesto en la sección anterior 3.5. Para ello se han definido una serie de valores de entrada que se muestran a continuación. k representa el número de componentes del servicio, es importante no confundirlo con el número de instancias, que viene dado por el vector N donde la posición i representa el número de instancias del componente i. En el vector D se establecen las demanadas de cada componente. *MaxCompInstances* es el número de instancias máximo por cada componente mientras que *MaxInstances* es el número máximo de instancias totales. Las constantes *SLA_UP* y *SLA_DOWN* definen el rango en el que se quiere que se encuentren los tiempos de respuesta de la aplicación, así, si el tiempo de respuesta es superior al *SLA_UP*, la configuración en ese momento está violando el SLA, por lo que el controlador deberá aumentar el número de instancias de los componentes, y lo mismo sucede en el caso de que el tiempo de respuesta sea demasiado rápido, por debajo del valor definido por *SLA_DOWN*, aunque en este caso en vez de aumentar, el controlador deberá reducir el número de instancias. Además para conseguir una mayor homogeneidad en las pruebas se ha decidido usar un generador de carga triangular tal que genera una carga de entrada tal y como aparece en la figura 3.4.

Datos de las pruebas:

```
k = 6;
D = [0.01 0.03 0.05 0.012 0.065 0.04];
N = [1 1 1 1 1 1];
MaxCompInstances = 10;
MaxInstances = k * MaxCompInstances;

SLA_UP = 1.0;
SLA_DOWN = 0.7;

load = cargaTriangular(80, i);
```

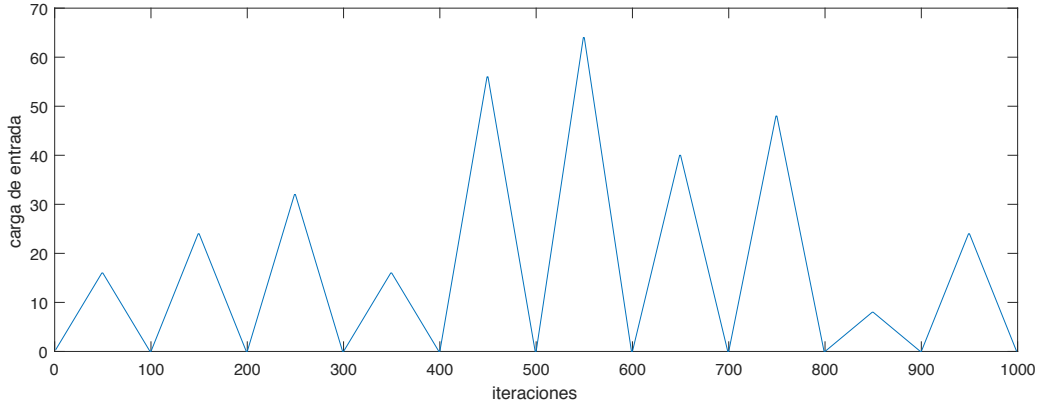

Figura 3.4: Carga de entrada al sistema

3.7.1. Estrategia Agresiva/Relajada

La estrategia Agresiva/Relajada es la más sencilla de las tres que se han propuesto en este trabajo. Consiste en aumentar de forma agresiva el número de instancias del servicio cuando se produce una violación del valor del SLA superior o SLA_UP, por ejemplo, para las pruebas experimentales que se muestran a continuación esa estrategia agresiva consiste en aumentar en una instancia cada componente del servicio, de esta forma, con los datos que se han visto anteriormente, cada vez que se viole el SLA_UP, el controlador añadirá k instancias al total de la configuración, haciendo que el coste aumente de forma rápida. Una vez que se han aumentado las instancias, como se ha realizado de forma agresiva es probable que se supere de sobra el número necesario para ese momento, y es ahí donde entra en juego la estrategia pasiva, que trata de ajustar el número de instancias reduciendolas poco a poco. Para estas pruebas la estrategia relajada supone quitar una instancia a un componente elegido de forma aleatoria. Esto se ve claramente en la figura 3.5 donde los saltos a la hora de aumentar las instancias son mucho más amplios que a la hora de reducirlas.

El número de violaciones del SLA medio que se produce con este método se situa en torno al 10 %, y los fallos que se producen en los clasificadores *Arow* no superan el 10 % tal y como se puede observar en las figuras 3.8 y 3.9

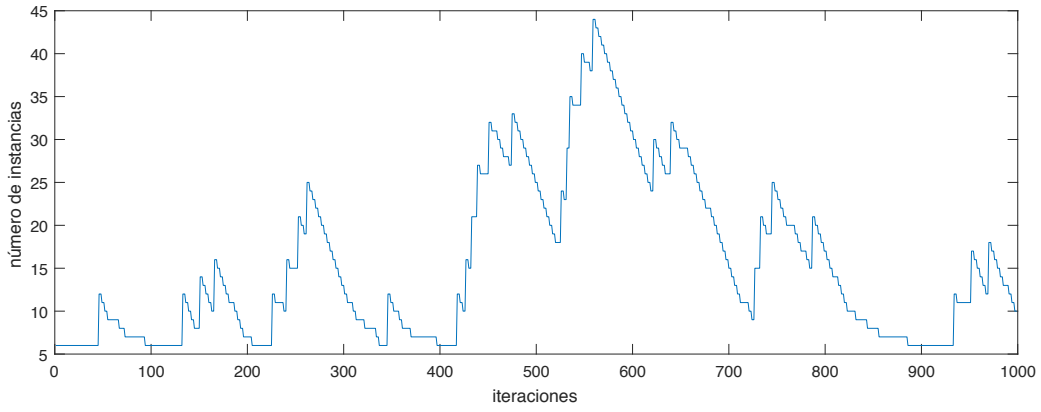

Figura 3.5: Coste de configuración del sistema

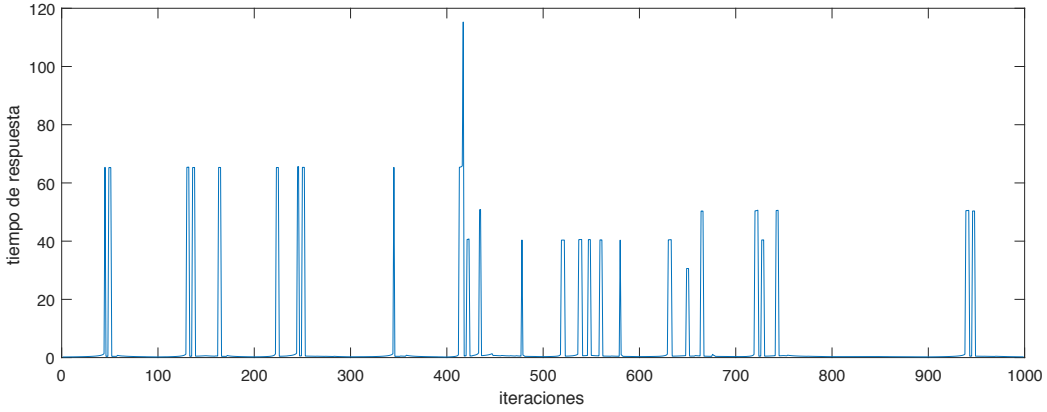

Figura 3.6: Tiempo de respuesta

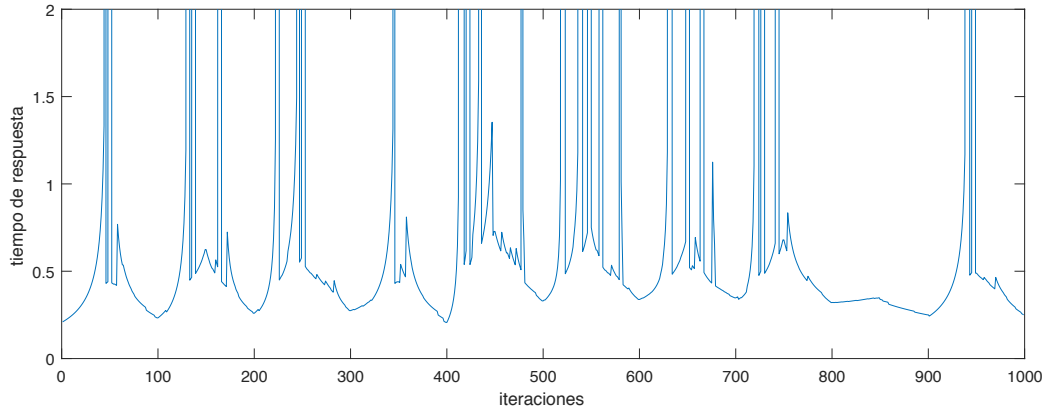

Figura 3.7: Detalle tiempo de respuesta

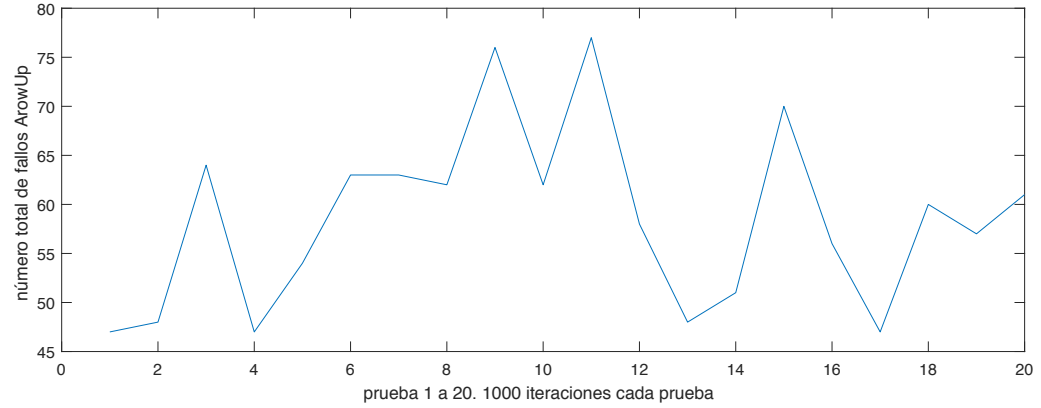

Figura 3.8: Fallos del clasificador ArowUp

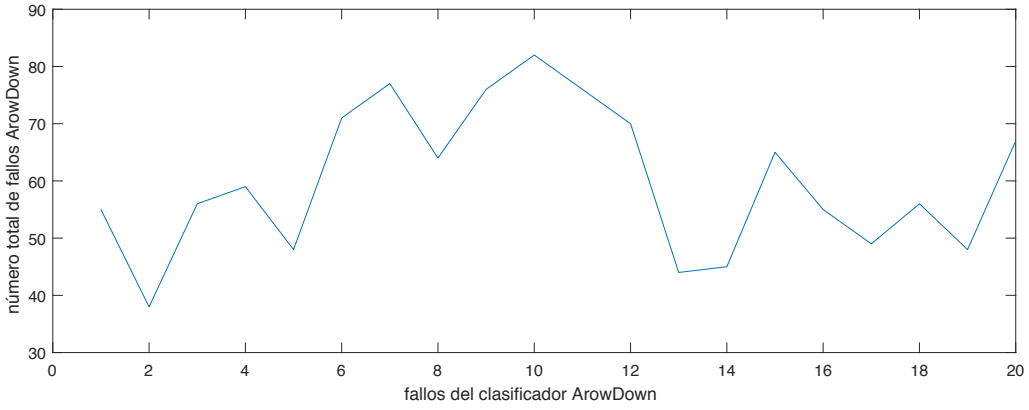

Figura 3.9: Fallos del clasificador ArowDown

3.7.2. Estrategia mediante Hill Climbing

A continuación se presentan los resultados obtenidos mediante el uso de la estrategia Hill Climbing, tanto cuando aciertan como cuando fallan los clasificadores *ArowUp* y *ArowDown*. Como se puede observar en las figuras 3.10 y 3.15 este método consigue ajustar con más precisión el número de instancias necesario en cada ocasión. En la figura 3.15 el método Hill Climbing viene dado por el color rojo, mientras que al método Agresivo/Relajado le corresponde el color azul.
Es fácil observar como la subida mediante Hill Climbing es mucho más escalonada, por lo que se alarga más en el tiempo, esto produce, debido a que existe un retardo a la hora de aplicar una configuración, que en el momento en el que se está aplicando esa configuración, sea probable que se vuelva a violar el SLA, de ahí que se observen unos picos tan grandes en el tiempo de respuesta tal y como muestran las figuras 3.17 3.12. No obstante, este hecho tiene una consecuencia positiva, y es que el coste de ejecución de la aplicación que utiliza Hill Climbing respecto a la aplicación que usa un método Agresivo/Relajado es menor.
Otro dato interesante nos lo da el número de violaciones del SLA medio con este tipo de estrategia, que para 4 ejecuciones de 20.000 iteraciones cada una, ronda el 35 %. Por último mediante las figuras 3.13 y 3.14 también podemos comprobar como aumenta el número de fallos en los clasificadores, hasta el punto de casi duplicar los fallos ocurridos en la estrategia Agresivo/Relajado.

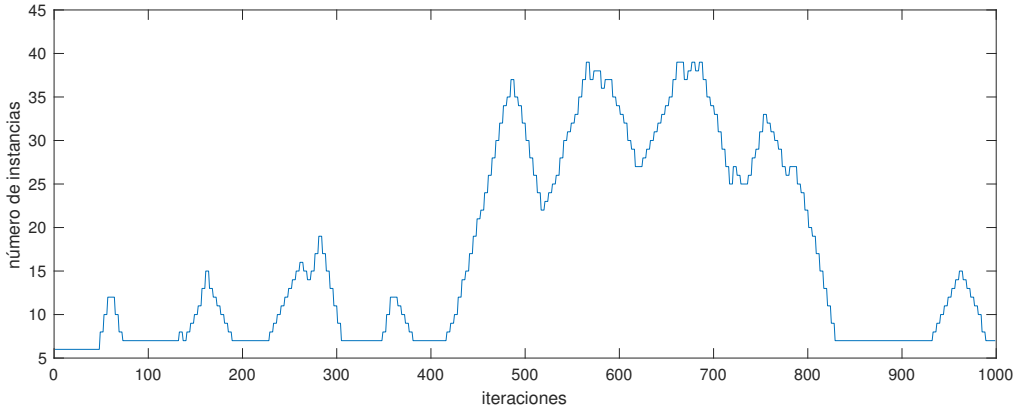

Figura 3.10: Coste de configuración del sistema usando Hill Climbing

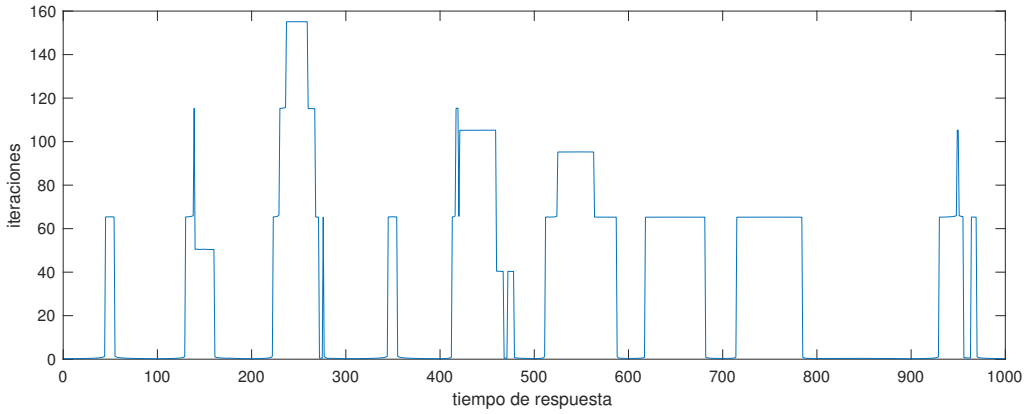

Figura 3.11: Tiempo de respuesta usando Hill Climbing

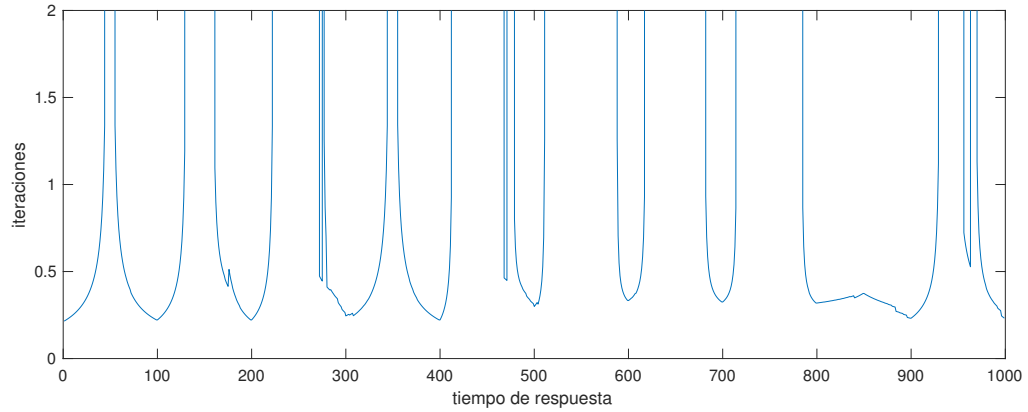

Figura 3.12: Detalle del tiempo de respuesta usando Hill Climbing

Figura 3.13: Fallos del clasificador ArowUp usando Hill Climbing

Figura 3.14: Fallos del clasificador ArowDown usando Hill Climbing

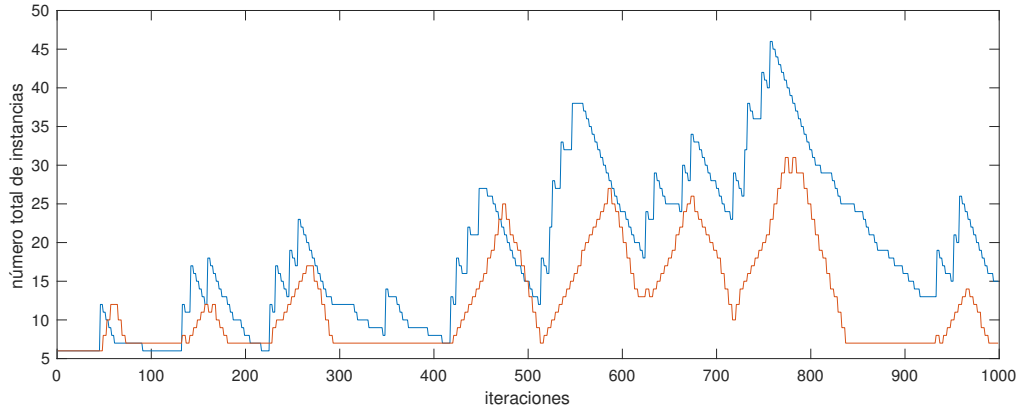

Figura 3.15: Coste de configuraciones comparativo

3.7.3. Estrategia Combinada

Con el objetivo de suavizar los resultados vistos en las estrategias anteriores, y obtener lo mejor de cada una se han combinado ambas estrategias, de forma que cuando la predicción del clasificador *Arow* se equivoca, se utiliza una estrategia agresiva en el caso de que sea necesario aumentar instancias, o relajada, en el caso en el que haya que reducir el número de instancias. No obstante, si el clasificador ha acertado, se opta por una estretegia mediante la búsqueda de la mejor configuración usando el algoritmo Hill Climibing modificado.

En la figura 3.21 se pueden observar los costes de las configuraciones de los tres métodos, en azul el método Agresivo/Relajado, en amarillo el método mediante Hill Climbing y en rojo la estrategia combinada. Es apreciable sobre todo en el caso en el que el controlador aumenta instancias, como hay momentos en los que la estrategia combinada utiliza ambos métodos. Por ejemplo, entre la iteración 100 y 200 se puede ver como comienza usando la estrategia Agresiva, no obstante entre la iteración 200 y 300 para la subida utiliza la estrategia Hill Climbing. En el caso de la reducción de instancias este fenónmeno es más dificil de apreciar, ya que con la estrategia relajada, solo se quita una instancia, y con la estrategia de Hill Climbing el máximo número de instancias permitido que se pueden quitar se ha definido en dos. A pesar de que haya momentos en los que esta estrategia combinada no funciona puesto que utiliza más instancias que incluso la estrategia Agresiva/Relajada hay casos como los que se dan a partir de la iteración 600 en los que ajusta mucho mejor el resultado. Esto también nos indica que conforme va aprendiendo el clasificador más certeros son los resultados del *Arow* y mejor se comporta el algoritmo de búsqueda Hill Climbing, de forma que el controlador va tendiendo a usar casi por completo este método y a utilizar la estrategia Agresiva/Relajada en momentos puntuales.

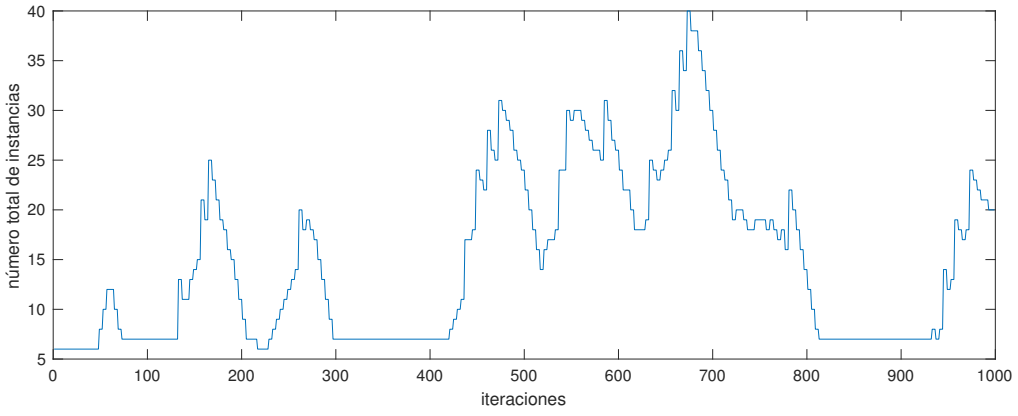

Figura 3.16: Coste de configuración del sistema usando la estrategia combinada

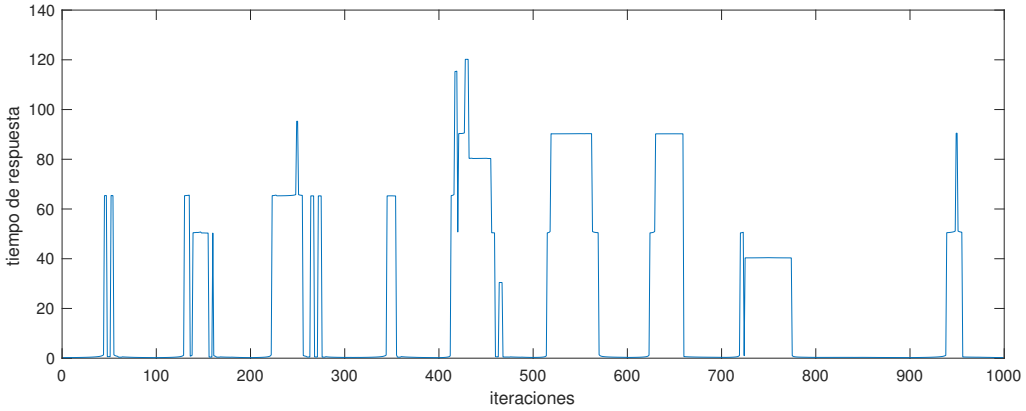

Figura 3.17: Tiempo de respuesta usando la estrategia combinada

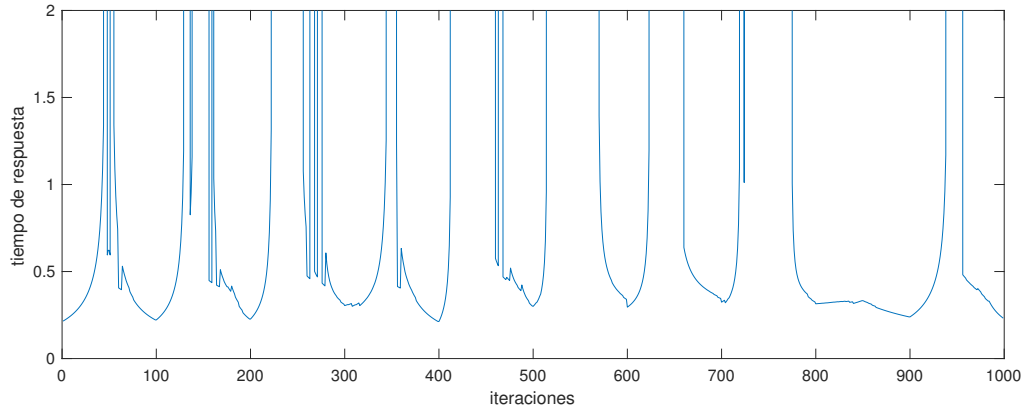

Figura 3.18: Detalle del tiempo de respuesta usando la estrategia Combinada

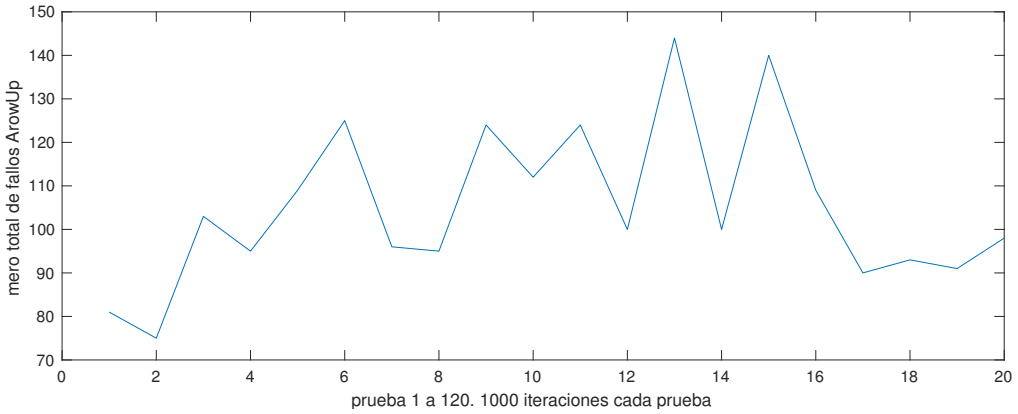

Figura 3.19: Fallos del clasificador ArowUp usando la estrategia Combinada

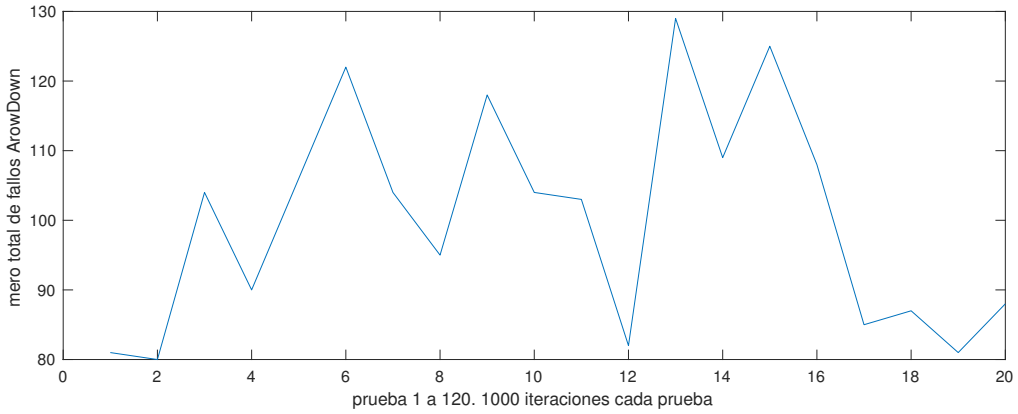

Figura 3.20: Fallos del clasificador ArowDown usando la estrategia Combinada

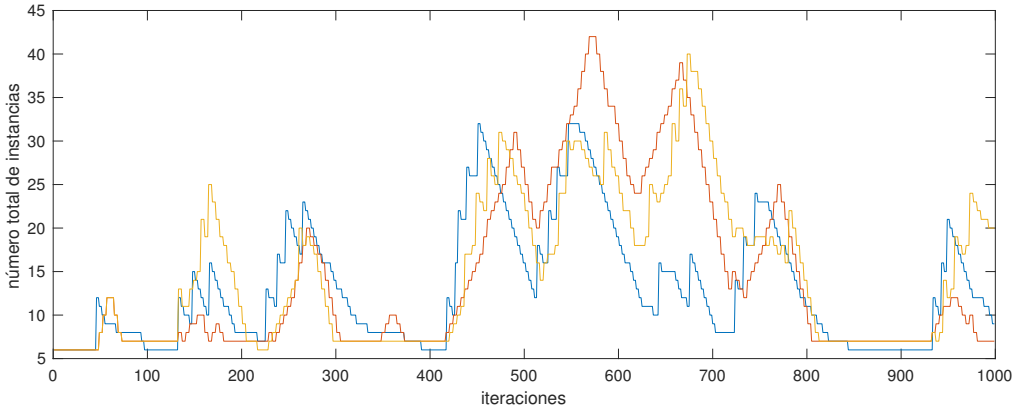

Figura 3.21: Coste de configuraciones comparativo

Capítulo 4

Conclusiones

Las pruebas experimentales nos han permitido concluir que mediante el uso de un sistema reactivo, se pueden alcanzar resultados aceptables a la hora de controlar la elasticidad de un servicio. Sus ventajas a la hora de aplicarlas en aplicaciones cloud, o basadas en microservicios son claras, Entre ellas destacan que no es necesario eleaborar complejos sistemas de métricas para conocer el estado de la aplicación, ni tampoco es necesario un sistema de aprendizaje previo, sino que el controlador, va aprendiendo mediante las técnicas de clasificación binaria vistas en secciones anteriores de un modo *On-line*. Todo ello repercute en una mayor facilidad a la hora del despligue y gestión de estas aplicaciones cloud, que al fin y al cabo también se traduce en un menor coste.

El sistema de clasificación binaria utilizado ha sido el AROW, y ha demostrado que a pesar de su sencillez consigue buenos resultados, especialmente en los casos en los que solamente aprende cuando falla su propia predicción. Además, como se ha visto es muy sencillo combinar distintos clasificadores AROW.

4.1. Investigación Abierta y Trabajo Futuro

Este trabajo abre nuevos caminos de investigación a la hora de controlar la elasticidad de servicios en aplicaciones Cloud:

- **Controladores predictivos basados en técnicas de clasificación binaria** Como se ha mencionado antes el controlador realizado para este trabajo es totalemnte reactivo. Con el fin de mejorar los resultados optenidos por este, se pueden plantear nuevos controladores predictivos tomando como base el desarrollado en este trabajo.

- **Controladores con otros algoritmos** Al igual que para este trabajo se ha utilizado el clasificador AROW, y una modifiación del del algoritmo Hill Climbing para las búsquedas es posible que otros den mejores resultados.

- **Aplicabilidad de este tipo de controladores en aplicaciones productivas** En este trabajo se ha tomado como base una aplicación ideal en la que todos sus componentes o microservicios son escalables, lo que dota a la aplicación de una alta escalabilidad, no obstante las aplicaciones reales, aún basadas en microservicios no tienen esta característas tan idóneas, por lo que aquí se abre una nueva línea de trabajo, que consiste en llevar estas ventajas a las aplicaciones que están en producción actualmente y que son por lo general mucho más complejas.

- **Inclusión de este tipo de controladores en servicios PaaS** Los PaaS actuales son las plataformas sobre las que se ejecutarán las aplicaciones del futuro. Es por ello que es importante poder trasladar las ventajas obtenidas por controladores como el presentado en este trabajo, de una forma sencilla, a los desarrolladores mediante su integración en la misma plataforma de ejecución.

Anexos

Apéndice A

Análisis MVA

A.1. Análisis operacional

Las redes de colas (Queuing Networks o QNs) se utilizan para evaluar el rendimiento de los sistemas. El modelo de redes de colas (QN model) de un sistam se utiliza para calcular las medidas de rendimiento que describen el rendimiento de ese sistema. Las medidas de rendimiento fundamentales incluyen el uso de recursos, la carga del sistema y los tiempos de respuesta. Existen cuatro leyes fundamentales aplicables a las QNs basadas en las métricas descritas en la siguiente tabla A.1:

Estas leyes nos permiten realizar análisis operacionales de diferentes sistemas

- **Ley de la Utilización (Utilization Law)**. La utilización es la parte del tiempo que el centro de servicio está ocupado.

$$U_i = S_i \times X_i \tag{A.1}$$

- **Ley de los FLujos Forzados (Forced Flow Law)**. Los flujos del sistema deben ser proporcionales.

$$X_i = V_i \times X_0 \tag{A.2}$$

- **Ley de Demanda de Servicio (Service Demand Law)**. La media total del tiempo de servicio requerido por un cliente.

$$D_i = V_i \times S_i = \frac{U_i}{X_0} \tag{A.3}$$

- **Ley de Little's (Little's Law)**. Es la ley que relaciona el número de usuarios en el sistema con el throughput y los tiempos de respuesta.

$$Q_i = X_i \times R_i \tag{A.4}$$
$$N = X_0 \times (Z + R_0) \tag{A.5}$$

A continución se explican los parmetros de entrada y salida del modelo QN

V_i	número medio de veces que un cliente visita el centro de servicio i
S_i	tiempo de servicio medio por cliente en el centro de servicio i
U_i	uso medio del centro de servicio i
X_i	throughput medio del centro de servicio i
D_i	demanda de servicio medio del centro de servicio i
R_i	tiemo de respuesta medio por operación del centro de servicio i
R'_i	tiempo de estancia medio del centro de servicio i
Q_i	longitud de la cola media del centro de servicio i
λ	tasa media de operaciones del cliente en el sistema
X_0	throughput del sistema
N	número medio de clientes en el sistema
R_0	tiempo de respuesta medio del sistema
Z	*think-time* medio de un usuario final

Tabla A.1: Lista de parámetros'

A.1.1. Entradas del modelo

Los centros de servicio representan recursos del sistema que pueden ser divididos en dos clases: *queuing* y *delay*. Los clientes en el centro de servicio de tipo *queuing* esperan para el uso del servicio. El tiempo empleado en este tipo de servicio se divide en dos componentes: El tiempo esperado en la cola del servicio y el tiempo gastado en el servicio. En un centro de servicio de tipo *delay*, no existe contención, es decir el tiempo de espera es cero.
La demada del servicio de un cliente en el centro de servicio i, D_i es un parmetro fundamental, representa la cantidad total de tiempo que requiere un cliente para ser servido en el centro de servicio. Bsicamente, es el número de visitas de clientes V_i multiplicado por la demanda del servicio por visita S_i, $D_i = V_i \times S_i$. La demanda total del servicio de un cliente en todos los centros de servicio es $D = \sum_{i=1}^{K} D_i$, donde K es el número de centros de servicio en el sistema.

- Las entradas de un sistema abierto son: (i) λ, la tasa de llegadas al sistema; (ii) K, el número de centros de servicio; y (iii) D_i, la demanda de servicio por cada centro de servicio i.
- Las entradas de un sistema cerrado son: (i) N, el número de clientes en el sistema; (ii) Z, el *thinking time*; (iii) K, el número de centros de servicio; y (iv) D_i, la demanda del servicio por cada centro de servicio i.

También es posible usar V_i y S_i como parmetros de entrada en vez de D_i.

A.1.2. Salidas del modelo

Las salidas comunes obtenidas después de evaluar el modelo QN son la utilización, el tiempo de estancia, el throughput y la longitud de la cola.

- El throughput es la tasa a la que los clientes se despachan del sistema (o de un centro de servicio) El throughput X_0 (una vez obtenido) puede relacionarse con el throughput de cada centro de servicio X_i mediante el uso de la Ley de Flujo Forzado (A.2): $X_i = V_i \times X_0$. Sin embargo, no es posible obtener esa relacion est parametrizado en términos de D_i.
- Tiempo de estancia R_0, Por ejemplo, la media del tiempo que transcurre entre la llegada de un cliente y su salida del sistema. Es simplementa la suma de tiempos de estancia en todos los centros de servicio: $R_0 = \sum_{i=1}^{K} R'_i$. Los tiempos de estancia se obtienen el los modelos QN de la evaluación del sistema, teniendo en cuenta tiempo de espera y la demanda del servicio.

- La utilización de un centro de servicio i, U_i, es la prorporción del tiempo que el centro de servicio est ocupado, Ej. $U_i = X_i \times S_i$, o por la Ley de Demanda de Servicio, $U_i = X_0 \times D_i$.
- La longitud de la cola en el centro de servicio i, Q_i, es la media del número de clientes en ese centro de servicio. El número de clientes en espera es simplemente, $Q_i - U_i$. Usando la Ley de Little $Q_i = X_i \times R_i$ y por la Ley del Flujo Forzado, $Q_i = X_0 \times R'_i$. En un sistema cerrado con N clientes, $N = X_0 \times (R_0 + Z)$. El término $X_0 \times R_0$ son los clientes recibiendo el servicio en el sistema, y $X_0 \times Z$ son los clientes *thinking* por ese servicio. En un sistema abierto, cuando el sistema alcanza un estado estable, ($X_0 = \lambda$), el número de clientes dentro del sistema es $Q = \lambda \times R_0$.

A.2. Técnica de solución del modelo cerrado

Esta sección presenta el algoritmo para el MVA (MEan Value Analysis) de un modelo QN cerrado con una única clase de clientes. Las demandas D_i son conocidas y son constatnes, p. e., tiempos de servicios son independientes de la carga (LI o Load Independent). La población y el *thinking time* del modeo son dados por los valores N y Z respectivamente. Definiendo $A_i(N)$ como el número medio de clientes vistos en el centro de servicio i en el mmomento de la llegada de un nuevo cliente a ese centro de servicio. A continuación, los centros de servicio sern denominados estaciones. En otras palabras, $A_i(N)$ es la longitud de la cola de llegadas a la estación i. Si $A_i(N)$ es conocida por cada i, entonces las siguientes ecuaciones resuelven el modelo QN:

$$\begin{aligned}
R'_i(N) &= D_i \text{ para una estación de tipo } \textit{delay} && \text{(A.6)}\\
R'_i(N) &= D_i(1 + A_i(N)) \text{ para una estación de tipo } \textit{queueing} && \text{(A.7)}\\
R_0(N) &= \sum_{i=1}^{K} R'_i(N) && \text{(A.8)}\\
X_0(N) &= \frac{N}{Z + R_0} && \text{(A.9)}\\
Q_i(N) &= X_0(N) R'_i(N) \text{ para cada estación de tipo } \textit{queueing} && \text{(A.10)}\\
U_i(N) &= X_0(N) D_i \text{ para cada estación de tipo } \textit{queueing} && \text{(A.11)}
\end{aligned}$$

with $1 \leq i \leq K$.

La técnica de solución exacta est basada en la computación de $A_i(N)$ usando el *Teorema de la llegada (Arrival Theorem)*. Para una red de colas de tipo cerrado *(closed, product-form queuing network)*, esto se define como

$$A_i(N) = Q_i(N - 1) \tag{A.12}$$

Una argumentación intuitiva de (A.12) es que en el instante en el que una nueva petición llega a una estación, no puede estar en su cola. p. e., no se puede 'ver' a sí mismo en la cola, Así, solo puede haber otros $N - 1$ clientes que podrían interferir con la nueva llegada. El número de estos en la cola es la media de la longitud de la cola en el centro cuando solo N -1 clientes estn en el sistema. La ecuación (A.12) permite un procedimiento iterativo mediante la definición de $Q_i(0) = 0$.

Algoritmo del MVA exacto (The exact MVA algorithm).

– –

input: N customer population; (D_i) demands; K number of stations.

output: $X_0(N)$; $R_0(N)$; $(R'_i(N))$ residence times; $(Q_i(N))$ queue lengths.

for $i = 1$ **to** K **do**
    $Q_i(0) = 0$
endfor

for $n = 1$ **to** N **do**
    //residence times, for each station i:
    for $i = 1$ **to** K **do**
        $R'_i(n) = D_i(1 + Q_i(n-1))$; (for a queuing station)
        $R'_i(n) = D_i$ (for a delay station)
    endfor
    //response time
    $R_0(n) = \sum_{i=1}^{K} R'_i(n)$;
    //throughput
    $X_0(n) = \dfrac{n}{(Z + R_0(n))}$;
    //queue length, for each station i:
    for $i = 1$ **to** K **do**
        $Q_i(n) = X_0(n)R'_i(n)$
    endfor
enfor

– –

La utilización de la estación i, $U_i(N)$, se calcula mediante (A.11). Observe que la complejidad de este algoritmo es $\mathcal{O}(NK)$.

A.3. Una solución aproximada

La técnica de solución exacta para el MVA est basada en la relación $A_i(N) = Q_i(N-1)$. Algunos autores han propuesto distintas aproximaciones para calcular $A_i(N)$, p. e., proveer una función $h[]$ tal que $A_i(N) \approx h[Q_i(N)]$. En particular, la aprocimación de Bard-Schweitzer es tal y como se presenta a continuación,

$$A_i(N) \approx h[Q_i(N)] = \frac{N-1}{N} \times Q_i(N) \tag{A.13}$$

Asumimos que N es grande, entonces la longitud de la cola en cada estación incrementa proporcionalmente de la siguiente forma:

$Q_i(N) \propto N$ y $Q_i(N-1) \propto N-1$. En el limite , para un valor grande de N, establece que:

$$\frac{Q_i(N-1)}{Q_i(N)} \approx \frac{N-1}{N}$$

Sustituyendo (A.13) en (A.7) produce la siguiente aproximación para el tiempo de residencia en la estación de tipo cola i:

$$R'_i(N) = D_i(1 + \frac{N-1}{N} \times Q_i(N)) \tag{A.14}$$

La proximación del algoritmo MVA se presenta a continuación.

Algoritmo MVA aproximado (An approximate MVA algorithm).

– –

input: N customer population; (D_i) demands; K number of stations; ε convergence value.
output: $X_0(N)$; $R_0(N)$; $(R'_i(N))$ residence times; $(Q_i(N))$ queue lengths.

// Estimate an initial queue length at each station i:
for $i = 1$ **to** K **do**
    $Q^e_i(N) = N/K$
endfor

repeat
    for $i = 1$ **to** K **do**
        $Q_i(N) = Q^e_i(N)$
    endfor;

    //residence times, for each station i:
    for $i = 1$ **to** K **do**
        $R'_i(N) = D_i(1 + \frac{N-1}{N}\, Q_i(N))$; (for a queuing station)
        $R'_i(N) = D_i$; (for a delay station)
    endfor;
    // response time
    $R_0(N) = \sum_{i=1}^{K} R'_i(N)$;
    //throughput
    $X_0(N) = \frac{N}{(Z + R_0(N))}$;
    //estimate the queue length, for each station i:
    for $i = 1$ **to** K **do**
        $Q^e_i(N) = X_0(N)R'_i(N)$
    endfor;

until $\max\{\frac{|Q^e_i(N) - Q_i(N)|}{Q^e_i(N)} | 1 \le i \le K\} < \varepsilon$

La solución aproximada reduce el tiempo de ejcución para un N grande en comparación con la solución exacta. Sin embargo, produce pequeas desviaciones en su precisión respecto al MVA exacto. El algoritmo de aproximación tiende a sobreestimar el tiempo de respuesta y a subestimar el thorughput del sistema.

A.4. Técnica de solución de modelo abierto

Esta sección presenta las ecuaciones que nos permiten resolver un modelo QN abierto. La tasa de llegadas al sistema λ, y la demanda del servicio, D_i, para cada centro de servicio i, $1 \le i \le K$, son los parmetros de entrada. El anlisis se realiza en estado estable, p. e., $X_0 = \lambda$.
La capacidad de procesamiento de un sistema abierto es la tasa de llegada en la que se satura. Para una tasa de llegadas muy alta, esto viene dado por:

$$\lambda_{sat} = \frac{1}{\max\{D_i\}} = \frac{1}{D_{bot}}$$

Como en general, $X_0 = \frac{U_i}{D_i} \le \frac{1}{D_i}$, la saturación ocurre en el caso the que una estación denominada *bot* satisface $U_{bot} \to 1$. En el anlisis que se presenta a continuación, se asume que $\lambda < \lambda_{sat}$.
En el caso abierto, dado que existe una posible cantidad infinita de clientes, eliminar un cliente del QN no tiene efecto. Esto significa que la longitud de la cola de llegadas en la estación i, $A_i(\lambda)$, es

simplemente la media de la logintud de la cola i, $Q_i(\lambda)$.

$$A_i(\lambda) = Q_i(\lambda) = X_i(\lambda) \times R_i(\lambda) = X_0(\lambda) \times R'_i(\lambda) = \lambda \times R'_i(\lambda) \tag{A.15}$$

La utilización por cada estación i es $U_i(\lambda) = X_0 \times D_i = \lambda \times D_i$.
Por operación, $R_i(\lambda) = S_i(1 + A_i(\lambda))$. Así, el tiempo de residencia es: $R'_i(\lambda) = D_i(1 + A_i(\lambda))$. En el modelo abierto, no se requiere un proceso iterativo, y el rendimiento de los resultados son calculados por fórmulas simples usando la utilización de cada estación.
Como, $\lambda \times R'_i(\lambda) = \lambda \times D_i(1 + A_i(\lambda))$, entonces $Q_i(\lambda) = U_i(\lambda)(1 + Q_i(\lambda))$. por lo tanto,

$$Q_i(\lambda) = \frac{U_i(\lambda)}{1 - U_i(\lambda)} \tag{A.16}$$

By previous (A.16), dividiendo por $1/\lambda$, se obtiene el tiendo de residencia.

$$R'_i(\lambda) = \frac{D_i}{1 - U_i(\lambda)} \tag{A.17}$$

La solución de modelo abierto se presenta a continuación:
The open model solution.
– –

input: λ arrival rate; (D_i) demands; K number of stations.
output:

Processing capacity: $\lambda_{sat} = \dfrac{1}{D_{max}}$

Throughput: $X_0 = \lambda$

Utilization, for each station i: $U_i(\lambda) = \lambda \times D_i$.

Residence time, for each station i:
$R'_i(\lambda) = \dfrac{D_i}{1 - U_i(\lambda)}$ (for a queuing station)
$R'_i(\lambda) = D_i$; (for a delay station)

Queue length, for each station i: $Q_i(\lambda) = \dfrac{U_i(\lambda)}{1 - U_i(\lambda)}$

Response time: $R_0(\lambda) = \sum_{i=1}^{K} R'_i(\lambda)$

Average number of customers in the system: $N = \lambda \times R_0(\lambda)$
– –

Se puede observar que la solución es estable si y solo si $\forall i : U_i(\lambda) < 1$.

Bibliografía

[1] Computación en la Nube, https://es.wikipedia.org/.

[2] Yasir Shoaib and Olivia Das. Performance-oriented Cloud Provisioning: Taxonomy and Survey. arXiv:1411.5077v1 [cs.DC] 19 Nov 2014.

[3] W. Zhang, et. al. Paas-Oriented performance modeling in Cloud Computing. 2012 IEEE 36th International Conference on Computer Software and Applications. pp. 395–404, 2012.

[4] N. Herbst, et. al. Elasticity in Cloud Computing. What It is, and What It is not. 10th International Conference on Autonomic Computing (ICAC 13). pp. 23–27, 2013.

[5] Prasad Jogalekar and Murray Woodside. Evaluating the Scalability of Distributed Systems. IEEE Transactions on Parallel and Distributed Systems, vol. 11, no. 6, pp. 589-603, june 2000.

[6] Chris Richardson. Introduction to Microservices. https://www.nginx.com/blog/introduction-to-microservices. 2016

[7] Tania Lorido Botran, et. al. A Review of Autoscaling Techniques for Elastic Applications in Cloud Environments. Journal of Grid Computing, vol. 12, pp. 559–592, 2014.

[8] An architectural blueprint for autonomic computing. Autonomic Computing White Paper. IBM, june 2005.

[9] C. Kan. DoCloud: An elastic cloud platform for web applications based on Docker. Proceedings of ICACT2016, pp. 482–487, 2016.

[10] D. A. Menasce et. al. Performance by design. Prentice-Hall, 2004.

[11] B. Urgaonkar et. al. An analytical model for multi-tier internet services and its applications. Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pp. 291–302, 2005.

[12] T. Zheng et. al. Performance Model Estimation and Tracking Using Optimal Filters. IEEE Transactions on Software Engineering, vol. 34, no. 3, pp. 391-406, 2008.

[13] Alessio Gambi et. al. Kriging Controllers for Cloud Applications. IEEE Internet Computing, pp.40–47, july 2013.

[14] Alessio Gambi. Kriging-based Self-Adaptive Controllers for the Cloud. Tesis doctoral. Faculty of Informatics of the Universitá della Svizzera Italiana, 2013.

[15] F. Muñoz-Escoí et. al. A Survey on Elasticity Management in PaaS Systems. Aceptado para su publicación, Computing, 2016.

[16] S. Malkowski et. al. Experimental Evaluation of N-tier Systems: Observation and Analysis of Multi-Bottlenecks. IEEE International Symposium on Workload Characterization,IISWC 2009.

[17] K. Crammer et al. Online Passive-Aggressive Algorithms. Journal of Machine Learning Research, vol. 7, pp. 551–585, 2006.

[18] K. Crammer et al. Adaptive regularization of weight vectors. Machine Learning, vol. 91(2), pp. 155–187, 2013.

[19] J. Liu, et al. Aggressive resource provisioing for ensuring QoS in virtualized environments. IEEE Transactions on Cloud Computing, vol. 3, no, 2, pp. 119–131, 2015.

[20] p. Jamshidi et al. Managing uncertainty in autonomic cloud elasticity controllers. IEEE Cloud Computing, vol. 16, pp. 50–60, 2016.

[21] A. Seidmann et al. Computerized closed queuing networks models of flexible manufacturing. Journal of Large Scale Systems, vol. 12, pp. 91–107, 1987.